AD-A069 539 SYSTEM DEVELOPMENT CORP SANTA MONICA CALIF F/G 9/2

SOFTWARE DEBUGGING METHODOLOGY. VOLUME I.(U)

APR 79 M FINFER, J FELLOWS, D CASEY F30602-77-C-0165

UNCLASSIFIED RADC-TR-79-57-VOL-1

1 OF 2

AD
A069539

LEVEL II

(12)

AD A069539

RADC-TR-79-57, Vol I (of three)
Final Technical Report
April 1979

SOFTWARE DEBUGGING METHODOLOGY

System Development Corporation

Marcia Finfer
Jon Fellows
Dan Casey

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

SOFTWARE DEBUGGING METHODOLOGY, VOL I

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

79 06 06 013

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-79-57, Vol I (of three) has been reviewed and is approved for publication.

APPROVED: Frank S LaMonica

FRANK S. LAMONICA
Project Engineer

APPROVED: Wendall C Bauman

WENDALL C. BAUMAN, COL, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-79-57, Vol I (of three)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SOFTWARE DEBUGGING METHODOLOGY		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report Sep 77 - Oct 78
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Marcia Finfer, Jon Fellows, Dan Casey		8. CONTRACT OR GRANT NUMBER(s) F30602-77-C-0165
9. PERFORMING ORGANIZATION NAME AND ADDRESS System Development Corporation, 2500 Colorado Avenue, Santa Monica CA 90406		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55810295
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIE) Griffiss AFB NY 13441		12. REPORT DATE April 1979
		13. NUMBER OF PAGES 170
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

18 6 Volume I. 9 19 TR-79-57-VOL-1 10 15 16 17 02 11 12 170 p.

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Frank S. Lamonica (ISIE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Software
Software Debugging
Software Testing
Debugging Tools
Debugging Techniques
Debugging Methodology

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A debugging study was conducted which surveyed current research being performed in the area of software debugging during integration-level testing. Particular emphasis was placed on assessing debugging tools and techniques which were applicable to embedded software developments. The purpose of the debugging study was to define a software debugging methodology applicable to diverse environments to be utilized during integration testing of system software. The results of the study are contained in three volumes. This volume presents (Cont'd)

Over

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

339 100

Item 20 (Cont'd)

a discussion of generic debugging information requirements which are used in the activities needed for integration-level debugging. Those activities are described in a debugging methodology which defines a logical progression of steps, or processes, deemed necessary to solve complex errors in system software. Many of the steps in the methodology are characterized as human thought processes and are, at best, rudimentarily described. Other processes are presented as support activities to these thought processes and include information identification, derivation, and collection. These processes require the use of debugging tools, which are generically described, for optimizing the efficiency of debugging activities. Considerations of the effect of different hardware/software environments on the debugging methodology and information requirements are also described. In addition, issues relating to a comprehensive debugging software support system are addressed to highlight the requirements to be considered in the procurement of debugging software.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

LIST OF FIGURES

LIST OF TABLES

EVALUATION

This effort involved the development of a methodology which defines the steps necessary to debug a reported software error, the information components required in each step, and software tools which can be used to support the task.

Its results are documented in a three-volume technical report. Volume I presents a discussion of generic debugging information requirements which are used in the activities needed for integration-level debugging. Volume II presents an application of the debugging methodology to three specific environments. Volume III presents the body of information used to derive the debugging methodology.

Use of the methodology is beneficial to the software development engineer who must evaluate and/or monitor proposals and software development efforts to determine the appropriateness of selected testing/debugging approaches, and to the debugging analyst who must be able to detect and resolve anomalous software performance in a variety of hardware/software environments for the testing of diverse software application problems.

This effort was performed under RADC TPO R5A/2.1 of which the objective is to develop software engineering tools and methods for use in the production and testing of Air Force software.

Frank S LaMonica
FRANK S. LAMONICA
Project Engineer

1. INTRODUCTION

This document presents a software debugging methodology to be used during the integration testing of software programs. The methodology describes, in generic terms, a structured process for the debugging analyst to follow in debugging, i.e., isolating and resolving an observed software error. In addition, it identifies the information structures he requires for the process and the tools he uses to obtain the information structures. As such, the debugging methodology presents a microcosmic view of the techniques of software engineering applied to one activity in the software development process. It is presented within the framework of the total development process because the information required for debugging encompasses all those activities which precede that activity.

1.1 PURPOSE

The purpose of this study is to define a software debugging methodology applicable to diverse environments to be utilized during integration testing of system software. The results of the study are contained in three volumes. Volume I, Software Debugging Methodology, presents a discussion of generic debugging information requirements which are used in the activities needed for integration-level debugging. Those activities are described in a debugging methodology which defines a logical progression of steps, or processes, deemed necessary to solve complex errors in system software. Many of the steps in the methodology are characterized as human thought processes and are, at best, rudimentarily described. Other processes are presented as support activities to these thought processes and include information identification, derivation, and collection. These processes require the use of debugging tools, which are generically described, for optimizing the efficiency of debugging activities. Considerations of the effect of different hardware/software environments on the debugging methodology and information requirements are also described. In addition, issues relating to a comprehensive debugging software support system are addressed to highlight the requirements to be considered in the procurement of debugging software. Volume II, Handbook for Debugging in the MULTICS/GCOS/RTM Environments, presents the application of the debugging methodology to three specific environments. Volume III, Literature and Site Survey Results, presents the body of information used to derive the debugging methodology.

1.2 CONCEPTS BASIC TO THE DEBUGGING STUDY

A number of concepts are described below which establish the framework on which the study was based. Other concepts used in the methodology are described in later sections within the context of their usage. Still other software development and software engineering concepts are described in Volume III, Literature and Site Survey Results. In addition, a glossary of terms used by the debugging methodology is appended to Volume I. The collection of terms and concepts used by the study is provided to establish the context in which software development occurs. Since this study is concerned with just one software development activity, it is being viewed in isolation of other development activities. A high quality and reliable software system reflects the structured, integrated and controlled process in which it was developed. However, since the debugging methodology assumes

a carefully controlled development process, it is important to establish some of the software engineering factors which are used in that process. In addition, since the methodology is generic in nature, the explication of commonly-used terms has been attempted by way of the various glossaries and definitions.

The concepts upon which this study is initially based include:

- <u>Testing</u> is the process by which the programmer ascertains that the program performs in such a manner as to meet its given functional/performance requirements.

- <u>Integration testing</u> is that testing applied to one or more separately compiled segments of code that have been combined, and when executed, interact with each other according to a design intended to solve or partially solve a function necessary to conform to specification.

- <u>Software specifications</u> are representations of the system at specific milestones in the development process. They are generically referred to as requirements specifications, design specifications and source program specifications. As such, the requirements specification is the statement of the problem, the design specification is the statement of a solution, and the source program specification is the statement of an implementation of the solution.

- <u>Back-tracking</u> is the process of examining the computational history of the software in reverse order to determine how an anomalous computation or condition was influenced by the computations and control structures previously executed.

- <u>Software debugging</u> is the process of isolating and resolving a software error (i.e., "bug") in one or more of the software specifications. Whereas testing demonstrates the error, debugging finds the location of the error, and (hopefully) points to a resolution. A "bug" in this study refers to a condition in a software specification which is causing software performance different from specified or required performance.

- <u>A test tool</u> is a hardware or software capability which can be used to demonstrate that the program exhibits behavior which meets its given functional performance requirements.

- <u>A debugging tool</u> is a hardware or software capability which is used to collect and display details of the software's operation before, during and after the occurrence of an error. It is used to help isolate the cause of an error and point to a solution.

- <u>Application software</u> is the end product of the software development effort and provides the capability which satisfies its user's requirements.

1.3 SCOPE

The debugging methodology presented in this report is one of the first attempts to provide a structured procedure for debugging software errors, in general, without regard to the specific application area, found during integration testing. As such it establishes a software engineering technique for one activity in one phase (test and integration) of the software development life cycle. However, the information structures required in debugging are developed in earlier phases of the software development life cycle (i.e., analysis, design, code and checkout phases). Descriptions of software engineering techniques that are used to develop, verify and maintain these information structures are not within the scope of the debugging methodology presented in this report. It must be realized that rigorous application of these techniques in earlier phases will substantially reduce the number of errors remaining in the software during integration testing.

The software debugging methodology is presented by way of a process model which was developed by partitioning a large subject area, in this case software debugging, into conceptually distinct processes; then defining the interactions and sequencing between these processes. Such an approach is necessarily limited by several factors. The first limitation is imposed by the essential arbitrariness of a given partitioning of the subject area. Any such partition is unlikely to be unique, and within a given partition some conceptual distinctions are simplified or ignored in order to support the choice of individual processes. Generally, different partitions will stress the simple expression of models which stress different conceptual aspects of the subject area. This problem is not unique to this study; indeed all engineering efforts are subject to the limitations of modeling arbitrariness. The motivation for creating this model is to aid the analyst in gaining insight into what happens during debugging and to understand how and when tools are used. In addition, its use may surface issues which will allow analysis of the effectiveness with which debugging can be performed. By partitioning the area of software debugging into smaller component processes, it becomes possible to subject each individual process to analysis of its information requirements and problems which can arise within that process. However, in no way does this model deny the validity of the quick insight an analyst may have in finding the cause of the software error. The model should be perceived as providing one methodology for debugging a logical flaw in software.

The debugging process model is primarily directed at detecting and resolving errors found in integration testing. This implies that smaller, isolated pieces of code or modules have been previously tested on a stand-alone basis. These modules are subsequently combined with other modules to determine the operational and functional compatibility of the combined set. Debugging, as addressed by the process model, begins with some evidence of program failure discovered during integration testing. Prior to the start of debugging, several events are presumed to have occurred which are of direct concern to the debugging process model, including:

- A structured and organized development process has occurred which produced, at a minimum, a requirements specification, a design specification, and a source program specification.
- An error has been identified and reported in integration testing.
- A report has been generated and entered into an error accounting system which identifies, at a minimum, the hardware/software configuration, test case and conditions, and functional capability under test.

Once the debugging process has isolated the cause of the error, the resolution of the error involves iteration of earlier design or coding activities, which may have widespread implications. The resolution indicated by the methodology does not include a description of the possible activities required in the implementation of the correction. Indeed, the resolution of complex software errors generally require an analysis of requirements, design, implementation, and documentation. It is felt, however, that the methodology should provide insight into the alternatives which exist for resolution of the error.

Two distinct audiences are addressed by this study. The first is the Air Force software development engineer who must evaluate and/or monitor proposals and software development efforts to determine the appropriateness of selected testing/debugging approaches. The second audience is the debugging analyst, who must be able to detect and resolve anomalous software performance in a variety of hardware/software environments for the testing of diverse software application problems. Both can benefit from the organized debugging approach offered; the explication of information structures to be generated, maintained and analyzed; the software support tools to be used by software development project personnel during debugging; and considerations of the impact of diverse hardware/software environments on debugging.

1.4 OVERVIEW OF THE DEBUGGING PROCESS MODEL

The model presented in this study and depicted in Figure 1-1 portrays software debugging in terms of a series of processes designed to clarify the steps necessary to debug a software error. Many of these steps are concerned with the human thought process, the description of which is beyond the scope of this investigation. However, consideration of the human thought process has brought into focus the information structures required by the debugging analyst to minimize the debugging effort and maximize the chances of success for problem isolation and resolution. (No assumption is made that all problems have solutions and/or are problems). It has also highlighted requirements for software tools needed to support the debugging process.

In essence, the debugging process model depicts the analytical and synthetic activities that enable the debugging analyst, starting from a report of anomalous program behavior, to find the initial source of an error. These analytic activities include the verification/duplication, localization, and execution

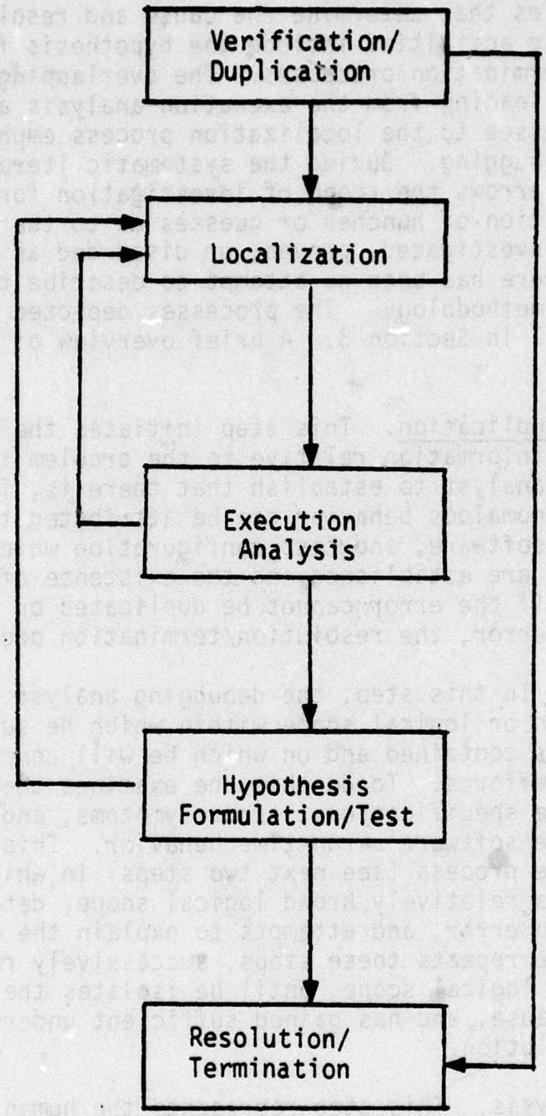


Figure 1-1. The Debugging Process Model

analysis processes. These are followed, and sometimes overlapped, by the synthetic activities that determine the cause and resolution of the problem. The synthetic activities include the hypothesis formulation/test and resolution/termination processes. The overlapping activities and the flow of direction leading from the execution analysis and hypothesis formulation/test processes to the localization process emphasize the iterative nature of debugging. During the systematic iteration of processes in which the analyst narrows the scope of investigation for the error cause, there is also an iteration of hunches or guesses as to the course and cause of an error that are investigated, proven, or discarded as part of the human thought processes. There has been no attempt to describe these intuitive iterations within the methodology. The processes depicted in Figure 1-1 are described in detail in Section 3. A brief overview of each process is presented below:

- Verification/Duplication. This step initiates the debugging process. All available information relative to the problem is reviewed by the debugging analyst to establish that there is, indeed, an error and that the anomalous behavior can be attributed to the software. The hardware, software, and test configuration which evoked the reported error are established and the existence of the problem is verified. If the error cannot be duplicated or if it is a man/machine error, the resolution/termination process is initiated.

- Localization. In this step, the debugging analyst defines the software domain or logical scope within which he suspects the cause of the error is contained and on which he will concentrate his investigative efforts. To do this, he examines the design and source language specifications, error symptoms, and additional evidence of the software's run-time behavior. This step is part of an iterative process (see next two steps) in which the analyst first selects a relatively broad logical scope, determines if it contains the error, and attempts to explain the error's cause. Then he repeats these steps, successively refining (i.e., narrowing) the logical scope, until he isolates the error's location and cause, and has gained sufficient understanding to recommend a solution.

- Execution Analysis. This step represents the human thought process in which the analyst analyzes the selected subset of the software to determine the relation of its function to the error. The process is characterized by the analyst making a series of assertions about the software's environment and the effect of its operation, and then testing those assertions, either mentally or by comparison with the results of the software's operation. This step, along with its predecessor and successor, is part of the iterative process of isolating the error's location and cause.

- <u>Hypothesis Formulation/Test</u>. In this step, the debugging analyst attempts to formulate a hypothesis that explains the software's anomalous behavior. He may find that he lacks sufficient understanding to do so, due to the broadness of the selected logical scope, in which case he iterates, returning to localization to narrow the logical scope. If he is successful in formulating a hypothesis, he tests its validity by examining relevant sections of the system descriptions or by modifying and retesting sections of the coded program. A validated hypothesis may lead the analyst directly to the resolution/termination process or it may require additional iterations to completely explain the error. An invalidated hypothesis requires a return to localization to further refine the logical scope.

- <u>Resolution/Termination</u>. This is the last step in the debugging process and serves as the transition between debugging and the other software development activities. The debugging analyst documents his conclusions about the error's cause and the actions required for problem resolution and then he initiates, but does not perform, those actions.

1.5 FURTHER WORK

Recommendations for further work are briefly described in Section 6 and cover the following topics:

- A study and correlation of error symptoms/error causes
- Integrated methodologies for tool/technique utilization within RADC environment
- Feasibility analysis/technical development plan: the NSW as a software engineering facility
- Use of assertions with program design notations
- Use of abstract data types with program design notation.

1.6 ORGANIZATION OF VOLUME

The remainder of this volume is organized as follows:

- Section 2 describes the external and internal information requirements needed by the debugging analyst in following the steps of the debugging methodology for isolating and resolving a software error.

- Section 3 presents the debugging methodology, via a process model, for isolating and resolving errors detected during integration testing and includes a description of the process relationships, information requirements, tool usage, assumptions made within each process, and probable sources of errors or problems.

- Section 4 examines the factors that determine generic hardware/ software environments and their impact on the debugging methodology.

- Section 5 addresses the procurement, design and development of a model debugging system which supports the use of the debugging process model.

- Section 6 contains a brief description of related areas that require further investigation.

- Appendix A describes the generic debugging tools that can support the generation, analysis, and maintenance of the information requirements of the debugging process model.

- Appendix B contains a glossary of terms used in this volume.

- Appendix C contains a selective bibliography that applies specifically to this volume. A more comprehensive bibliography is contained in Volume III.

2. DEBUGGING INFORMATION REQUIREMENTS

This section describes the information components required by the debugging analyst in isolating and resolving an observed software error. It identifies the information components, the relationship between the components and the relevance of the information and its representations to the debugging process.

Debugging, again, is the process of isolating the cause of a logical flaw in one or more of the various system specifications used to convert a desired capability into an executable implementation. The various specifications are referred to generically and include source program specification, design specification, and requirements specification. They may each consist of other information which, when combined with operational evidence, collectively contain the information required for debugging the errors they demonstrate. Operational evidence refers to that information obtained from software execution which demonstrates that actual program behavior is not consistent with what is required.

While debugging activities occur in several phases of the software development life cycle, this study focuses upon those debugging activities that take place during integration testing, and on the information requirements needed for that activity. It does not include a description of the methods used to develop and maintain the information that is generated in the activities preceding integration phase testing/debugging. Some of these considerations are discussed in Section 4, Impact of Environments on Debugging.

The debugging methodology presented in this report is generic in nature and should be applicable to debugging associated with integration phase testing of most application software. Since this testing is designed to verify the correctness of the software components' interfaces (with other software components, hardware, man-machine) and performance criteria, the debugging analyst must have the following categories of information available in order to determine which specification is flawed:

- Application System Descriptions. This category of information is described in Section 2.1 and includes:
  - A representation of the problem statement as specified in the requirements specification (e.g., B5 - Development (Part I) Specification).
  - A representation of the problem solution by way of a specific design specification (e.g., C5 - Development (Part II) Specification)
  - A representation of the implementation of the problem solution (e.g., the source program listing).
- Abstract Machine Descriptions. This category of information is described in Section 2.2 and includes the users manuals associated with the abstract machine (i.e., the combination of specific hardware and associated support software) which is being used for the implementation of the problem solution. Most often, the abstract machine used by the programmer is the higher order language (HOL) and associated compiler.

- Run-time Information. This category of information is described in Section 2.3 and includes:

  - The software discrepancy report and error symptoms produced during integration testing which indicates an anomalous condition in the hardware and/or software configuration when executed with a given set of input conditions. Additional error symptoms may be discovered throughout the debugging process.

  - The information relating to the internal state of the application software and abstract machine at a given point in the program's execution during the processing of a given set of input conditions.

- Derived Debugging Information. That information, described in Section 2.4, which is generated in each instance of debugging and is necessary for the isolation and resolution of the software error. It includes:

  - The logical scope (i.e., boundaries) within the source program specification, and/or related design and requirements specifications which contains the error. The logical scope is investigated by the debugging analyst, and refined in the debugging process until the error cause is isolated.

  - The assertions that the debugging analyst must make and test about the expected conditions (e.g., control paths, data variable values) that exist at a given point in the program's execution in response to the input data defined in a given test case. The validity of these assertions are determined by comparing the assertions with actual values. The results of the comparison guides the analyst in the refinement of the logical scope and the determination of which software component(s) contain the cause of the error.

  - The hypothesis that can be formulated and tested as an explanation for an invalidated assertion when the debugging analyst has sufficiently narrowed the logical scope. This explanation of the difference between asserted and actual values is (hopefully) the cause of the software failure.

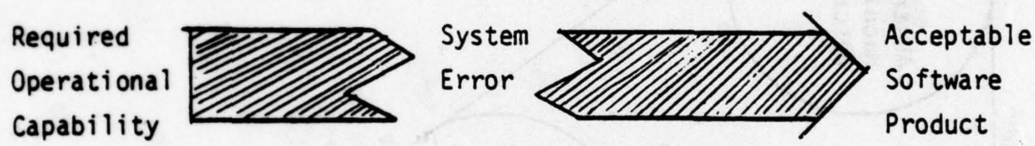
- Management Information. This category of information is briefly described in Section 2.5 and includes estimated and actual budget, schedule, and performance data which is used to decide the amount of resources that should be expended to isolate and resolve a specific software error, when such decisions are necessary.

The application system descriptions, abstract machine descriptions, management information, and the initial run-time information are generated outside of the debugging process. The generation, analysis and maintenance of this data is vital to the total software development process, but a discussion of these activities is outside the scope of this study. The derived debugging data and most of the run-time information are generated as part of the debugging

process. The generation and analysis of derived debugging data are an intrinsic part of the human thought processes that go on throughout debugging. Because of this close relationship, the discussion of the derived debugging information is contained partially in this section and partially in Section 3. The generation and analysis of run-time data are part of the tool using processes of debugging. A description of generic tools used in debugging is found in Appendix A, Generic Debugging Tool Requirements.

The debugging analyst relates these information components to each other through a complex process (debugging) which is rather rudimentarily portrayed in the debugging process model described in Section 3. The various information components which are used and/or generated during debugging must be combined to reflect the operation of an implementation of a problem solution to a given set of input conditions on a given abstract machine. Stated another way, in their entirety, the information components define the relationships between the abstract machine, the executable programs, and a design which is thought to be responsive to a set of functional performance requirements.

A system error can be considered as a gap between the required operational capability and an acceptable software product, graphically depicted as follows:



In order to first locate and then bridge this gap, the debugging analyst must conceptually integrate the relationship between the application system description, the abstract machine description and the run-time information. The result is the derived debugging information which is used to locate and bridge the gap (i.e., find the error). The error can be in the statement of requirements (the problem statement), the design (the problem solution), the implementation of the design (the source program) and the data representing the execution of the implementation on an abstract machine (the run-time data). This relationship is attempted in Figure 2-1.

Each of the information components presents the debugging analyst with quite different views of the software system and its functional/performance characteristics. As such, they present a unique view of the software development process, as follows:

- The software requirements specification represents a statement of software functional/performance capabilities required by and agreed to by the procurer and the developer, respectively.

- The design specification represents a design for a specific implementation of the requirements which is response to the acceptance criteria mutually agreed upon by the procurer and developer.

- The source program specification represents an implementation of a design based upon a specific abstract machine.

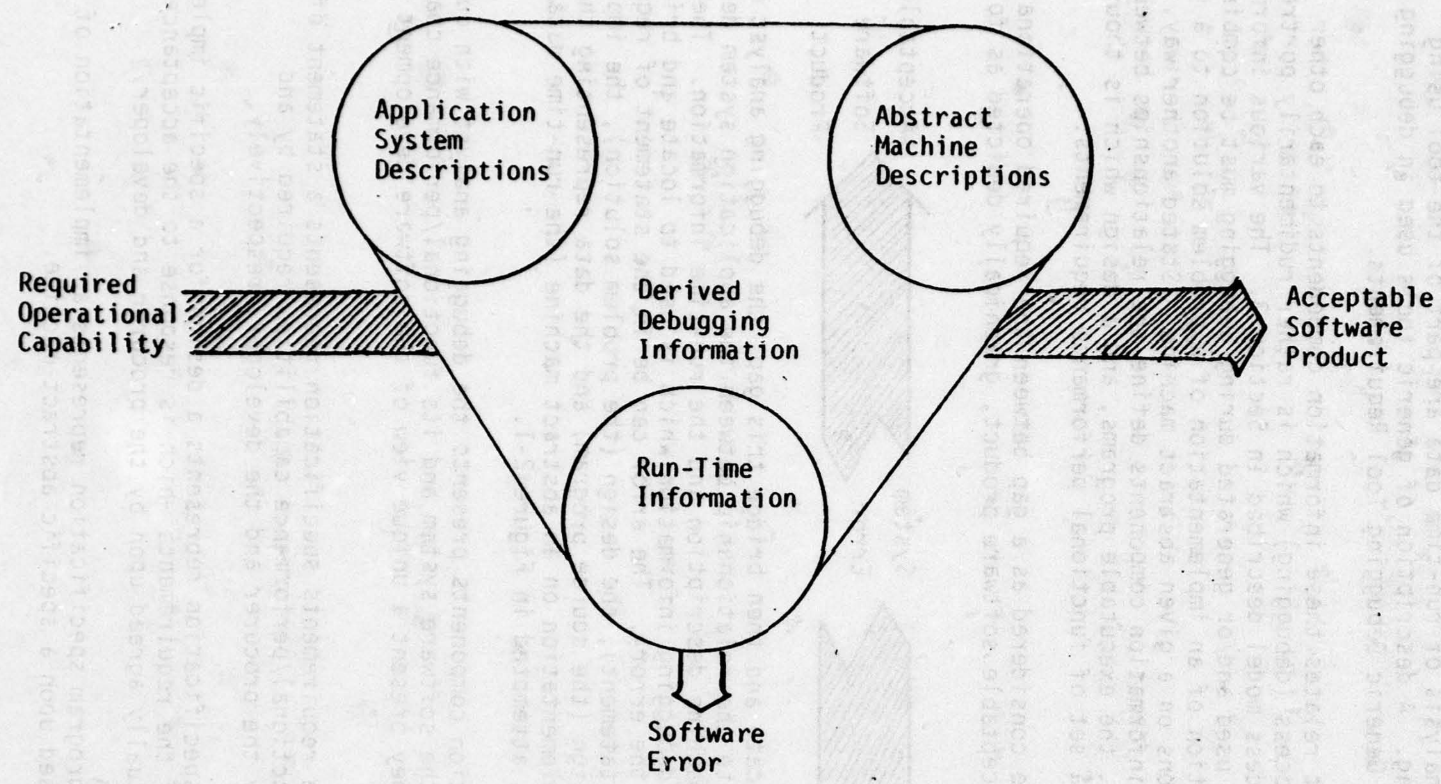


Figure 2-1. Information Component Relationship

- The abstract machine descriptions provide a description of the context for the implementation.

- The run-time information provides the information by which to determine if the specific implementation performs as specified (within tolerance limits), and will be accepted by the procurer.

However, all of the above requires that errors in software implementation can be isolated and resolved (via derived debugging information) to perform as specified. *It is the derived debugging information, then, which resolves* discrepancies between required and actual software performance in the various specifications of the implementation.

2.1 APPLICATION SYSTEM DESCRIPTIONS

The application system descriptions are those documents, or specifications, produced in the development of the software product. (The activities and events of the software development life cycle are discussed in Volume III, Section 1.3.2, and will not be discussed in detail in this volume.) The application system descriptions for a computer program configuration item (CPCI) are essential to the debugging methodology. Their relationship to the life cycle is depicted in Figure 2-2.

Integration testing and debugging activities for a CPCI are seen to continue into the System Integration and Test and System Operations phases of the software development life cycle (as shown on Figure 2-2) because of *engineering change* proposals (ECPs). An engineering change proposal is the term used to include the engineering change to the system and the documentation by which the change is described. An engineering change is a modification to the configuration of an approved CPCI. ECPs must be included in the application system description because a large majority of errors appearing in production software are introduced as unintended side effects of post-release program modifications.

The application system descriptions are generically described and referred to. They include the requirements specification, design specification, and source program specification. This does not exclude the possibility, however, that each of these documents may consist of several other documents. In software developments governed by military regulations, specifications and standards, there are additional documents that should be included in the generic application system specifications. In this study, the documents have been grouped together in a manner that demonstrate their use in debugging. Table 2-1 presents this correspondence and indicates the additional documents included with each generic application system description. A brief description of these documents follows.

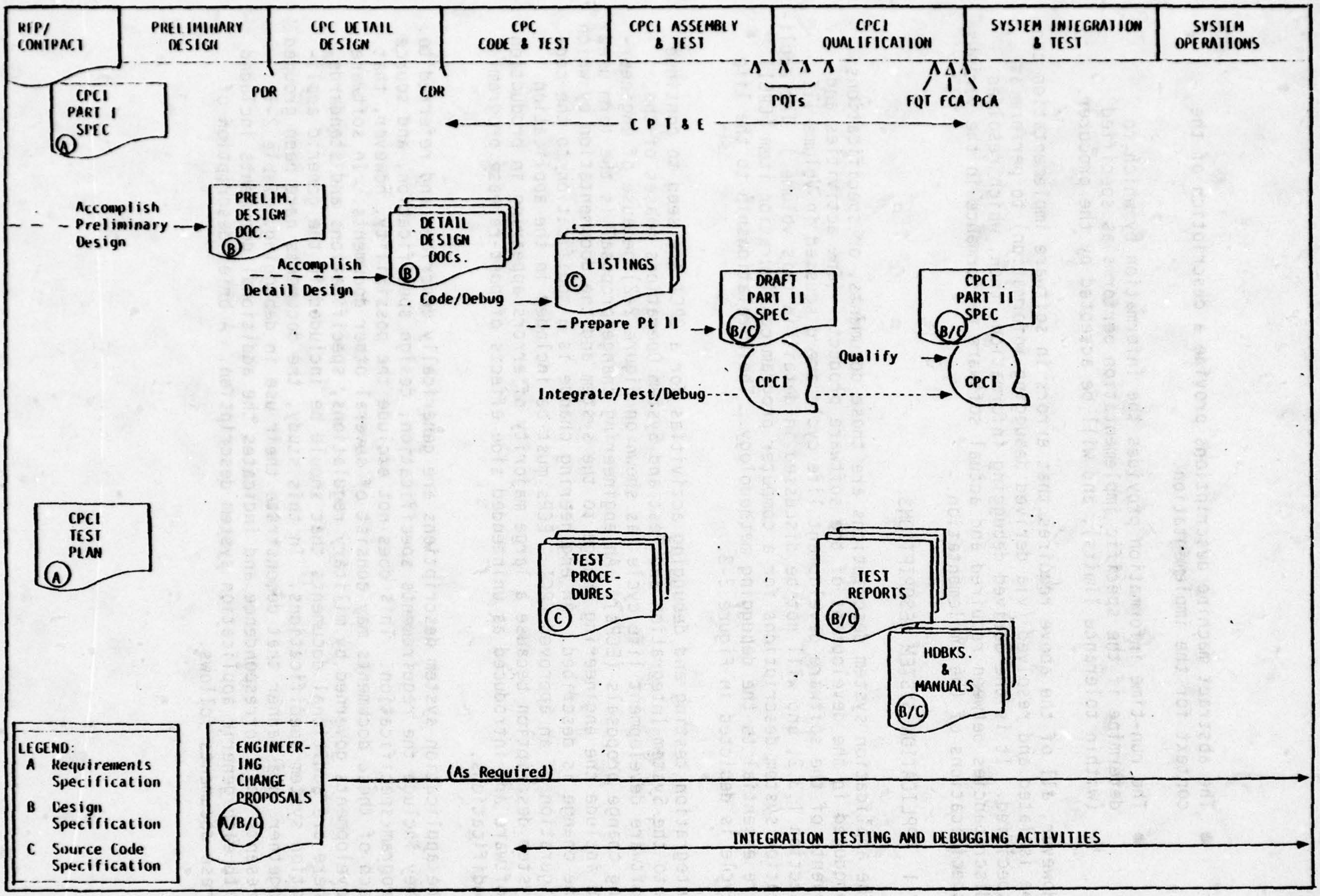


Figure 2-2. Model for Full-Scale Development Phase Relationships with Integration Testing and Debugging Activities

Table 2-1.

Application System Descriptions	Military Regulations, Specifications and Standards	
	Name of Document	Governing Publication
Requirements Specifications	Development (Part I) Specification	MIL-STD-490 (B5)
	CPCI Test Plans	USAF DID DT-T-3703A
Design Specification	Product (Part II) Specification	MIL-STD-490 (C5)
	Preliminary Design Document	USAF DID DI-T-3120A
	Detail Design Document	USAF DID DI-T-3120A
	Interface Control Documents	
Source Program Specification	Product (Part II) Specification	MIL-STD-490 (C5)
	Test Procedures	USAF DID DT-T-3703A
	Version Description Document	USAF DID DI-E-3121/M
	User Manuals	USAF DID DI-M-3409
		USAF DID DI-M-3410

MIL-STD-490 identifies and specifies the practices governing the preparation of the B5-Computer Program Development (Part I) Specification which defines it as containing the requirements for CPCI performance, design and interface. MIL-STD-490 also identifies and specifies the practices governing the preparation of the C5-Computer Program Product (Part II) Specification which documents the technical design and coding of the CPCI. This document, in draft format, is available during integration-level testing and debugging and it provides the required design specification information. Other documents important to integration-level testing and debugging are the CPCI Test Plan and Test Procedures. The CPCI Test Plan augments the acceptance criteria given in the Computer Program Development (Part I) Specification for each performance requirement. It identifies the formal tests that are to be conducted, their objectives, schedules and support requirements. The CPCI Test Procedures specify the test objectives of detailed test scenarios, including test events, inputs, expected results, and methods of test data recording and analysis. The use of the CPCI Test Plans/Procedures in the debugging methodology is described in Section 3.1, Overview of Integration-Level Debugging.

Methods for generating and maintaining the application system descriptions are assumed to exist within the development methodology. The overall quality of the application system description is a direct factor in the reliability of the operational system and an indirect factor to the efficiency and eventual success of the debugging analyst. Since the debugging analyst may be unfamiliar with the descriptions, the use of techniques to enhance his comprehension are highly desirable. These techniques include:

- Automated requirements and design specification tools that are used to describe and record such specifications in a machine-processable form. A variety of reports can be produced which aid the analysis of the specification. An example of such a tool is the "computer-aided" Design and Specification Analysis Tool (CADSAT), also referred to as URL/URA.

- Modern programming practices that promote the development of a design and its specification that is easy to read and understand.

- Structured programming constructs that promote the development of source language specifications that are easy to read and understand.

The use of these techniques are not required for the application of the debugging methodology described in Section 3, but they make it (and, indeed, any debugging methodology) more effective. These issues are addressed in more detail in Section 4.

Current software engineering practices advocate that requirements statements, design specifications, and source program specifications should be developed as hierarchical structures. The process of developing a hierarchy of increasingly less abstract concepts is called hierarchical decomposition. Although very few systems are developed which exhibit hierarchical structure in the full technical sense of the term, the concept of a hierarchical structure is useful in describing the representations of the system at each phase in the software development process. Given that the application system descriptions represent the system at three specific levels of abstraction, the debugging analyst must have available the documentation representing these three levels in his investigation of a logical flaw. These levels of system abstraction are represented by that level which is the problem statement (i.e., the requirements specification); that level which is a problem solution (i.e., a design specification); and that level which implements a design (i.e., source program specification). A logical flaw is an error existing in one of the three representations which causes results different than specified. The flaws between each representation are those errors that occur when an abstract concept is translated into a concrete solution. The flaws within each representation are errors that can occur in the translation of acceptable input conditions and values to specified output actions and values. The set of input, processing, and output specifications represent the required performance of the system at specific levels of abstraction.

Both informal and state-of-the-art formal verification research efforts are considering methods for determining the correctness of specifications within and between levels of abstraction for hierarchical system developments. Figure 2-3 presents an example of hierarchical system development with identification of the steps needed for translation and verification of specification. (The "I→O" symbol relates to the transformation of acceptable inputs (I) to specified output values (O) within a representation). Ideally, the verification processes depicted are performed during the hierarchical decomposition process. If indeed these methods prove unconditionally successful, the need for integration testing/debugging may diminish, if not disappear completely.

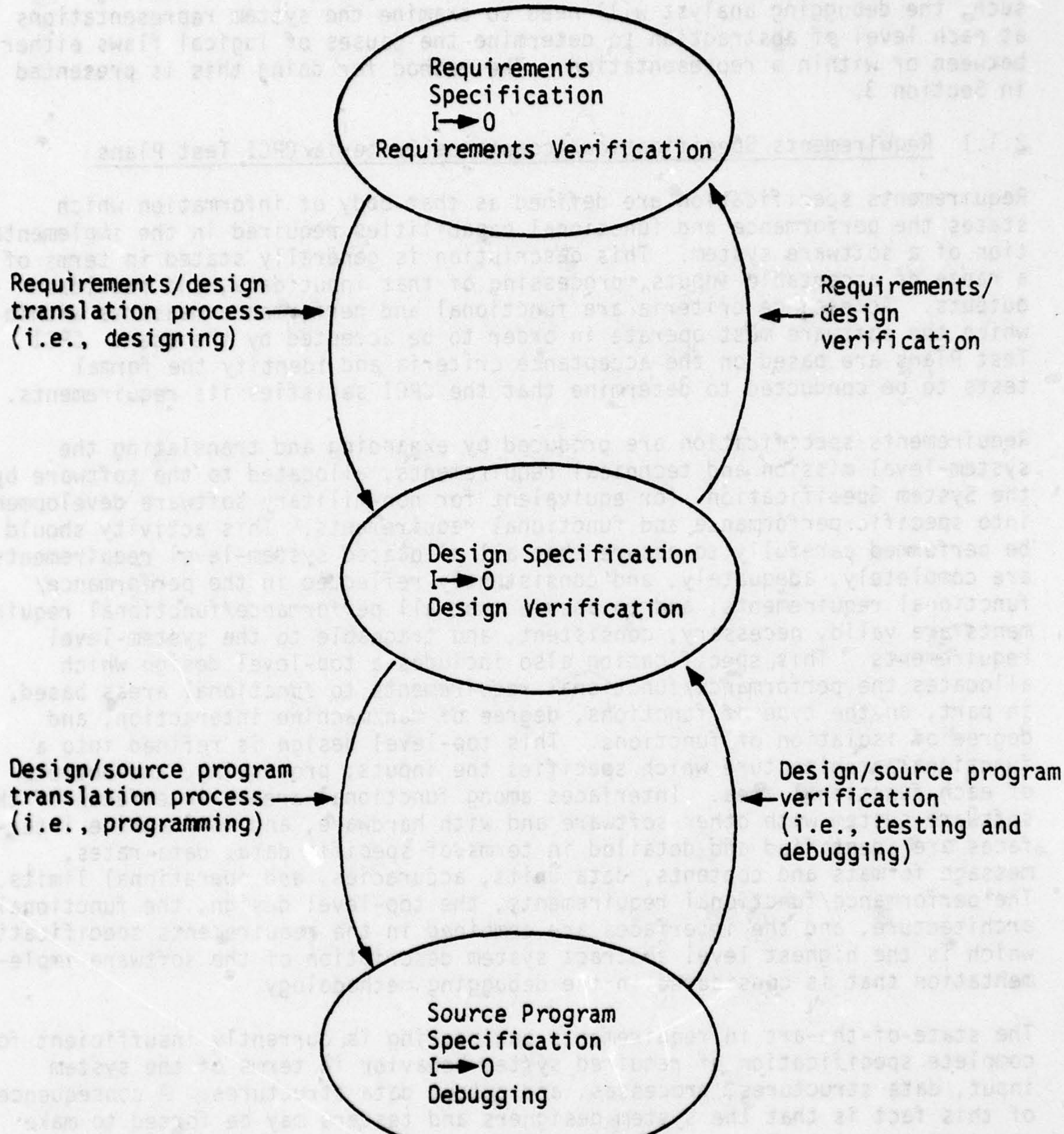


Figure 2-3. Hierarchical System Development

However, the debugging methodology presented in this study assumes the imperfection of at least some of the activities required in verification of the application system specifications preceding integration phase testing. As such, the debugging analyst will need to examine the system representations at each level of abstraction to determine the causes of logical flaws either between or within a representation. The method for doing this is presented in Section 3.

2.1.1 Requirements Specification/Acceptance Criteria/CPCI Test Plans

Requirements specification are defined as that body of information which states the performance and functional capabilities required in the implementation of a software system. This description is generally stated in terms of a range of acceptable inputs, processing of that input data, and resultant outputs. Acceptance criteria are functional and performance measures within which the software must operate in order to be accepted by the user. CPCI Test Plans are based on the acceptance criteria and identify the formal tests to be conducted to determine that the CPCI satisfies its requirements.

Requirements specification are produced by expanding and translating the system-level mission and technical requirements, allocated to the software by the System Specification (or equivalent for non-military software developments) into specific performance and functional requirements. This activity should be performed carefully to assure that all allocated system-level requirements are completely, adequately, and consistently reflected in the performance/functional requirements; and to assure that all performance/functional requirements are valid, necessary, consistent, and traceable to the system-level requirements. This specification also includes a top-level design which allocates the performance/functional requirements to functional areas based, in part, on the type of functions, degree of man/machine interaction, and degree of isolation of functions. This top-level design is refined into a functional architecture which specifies the inputs, processing, and outputs of each functional area. Interfaces among functional areas, interfaces of the software system with other software and with hardware, and man/machine interfaces are identified and detailed in terms of specific data, data rates, message formats and contents, data units, accuracies, and operational limits. The performance/functional requirements, the top-level design, the functional architecture, and the interfaces are combined in the requirements specification, which is the highest level abstract system description of the software implementation that is considered in the debugging methodology.

The state-of-the-art in requirements engineering is currently insufficient for complete specification of required system behavior in terms of the system input, data structures, processes, and output data structures. A consequence of this fact is that the system designers and testers may be forced to make assumptions about what constitutes correct system performance which can lead to an incorrect system design, and ultimately, to incorrect system behavior. At the onset of the debugging process, however, it is assumed that the requirements correctly specify system performance, requiring that the analyst look for the cause of a system failure in the implementation and/or transformation of requirements. If the cause of system failure cannot be found in the transformation of requirements to design, or from design to source language specification, the debugging analyst may have to trace the cause of the observed error back to the requirements specification. The debugging analyst

will then need to examine it for errors in its internal correctness, completeness, and consistency, or for errors in formulating its acceptance criteria. In some cases, it may be necessary to refer to that body of documentation which supported the allocation of system functions to software and upon which the requirements analysis and specification process is based. This body of information is assumed to exist in Air Force procurements but is not addressed further in this report. It includes the following specifications:

- System Specification to describe the required behavior on a system level.

- Segment Specification, used in large systems, to describe the required behavior for a logical segment of a system; for example, the ground segment or spaceborne segment of a satellite system.

- Group Specification, used in large systems, to describe the required behavior of an aggregate of configuration items which corresponds to a major functional capability of a segment.

Acceptance criteria are of concern to the debugging methodology because they provide a statement of required performance generally within specified acceptance tolerances. This information is necessary for comparisons with actual software behavior within the debugging process. Ideally, the required performance of the software should be defined in sufficient detail for direct use as an acceptance criteria (e.g., as quantified values or go/no-go conditions). This permits the test reporting process to be objective where differences between the actual and required software performance are reported. It also allows the debugging processes to evaluate actual and required behavior in the same terms. In addition, the acceptance criteria should be designated for each performance requirement in such a way as to provide ready associations with the appropriate software component. Methods of verifying that the requirement has been met may include review of analytical data, examination of displays, demonstration tests, reviews of test data, inspection of the program, etc. Identification and association of performance requirements, verification requirements, and methods are generally included in the specification by a traceability matrix which identifies each requirement, the level(s) of testing to be employed in verifying the requirement, and the verification method to be used. Any changes in the requirements specification due to ECP's may result in changes in this matrix, and ultimately, to the test plans/procedures, as well as changes to the design and source language specifications.

The CPCI Test Plan, based on the acceptance criteria, formulates the general test methods and evaluation criteria to be used to demonstrate that the software meets its performance and functional requirements. It identifies formal tests to be conducted (PQTs and FQTs), their objectives, relative schedules, and support requirements. The CPCI Test Plan should address the following topics:

- Identify the performance requirement, function or interface to be tested.

- Identify the method to be used to verify the requirement.

- Specify the test tools, equipment configuration and organizational roles.

- Identify types and sources of acceptance criteria and tolerances (where applicable).

- Specify test data requirements.

- Schedule the test activities.

2.1.2 Design Specifications

The high-level system design and architecture included as part of the requirements specification is further refined in the design phase. This includes detailing the input, intermediate, and output processing steps of each program module. In addition, as each detailed design evolves, it is analyzed for its suitability, consistency, and completeness, for its compatibility with the functional architecture, and its feasibility for implementation. During the detailed design, the initial data base definition is updated to reflect any newly identified data structures. The results of the entire design effort are then combined to produce the design specification.

The design specification contains both the high-level design descriptions relating functional capability to system architecture and the detailed module design description reflecting input, processing, and output of each functional capability. It also contains a description of the functional requirements of each sub-module level. Quality characteristics of the design specification that impact the debugging process reflect both the design phase methodology used and the representation chosen for the resultant design specification. For example, the use of an hierarchical decomposition, structured programming methodology to successively refine a software system description, starting from the high-level design and continuing until a very detailed, program-level description is reached, will result in a design specification in which each individual program module represents an individual software function and the interdependence between modules is minimized to achieve simplicity, flexibility, and overall reliability. This type of design simplifies the task of isolating the cause of an error in debugging. If the design of modules at each level describes the data affecting and affected by the module in implementing a function, the debugging analyst can use this information to localize the scope of the error, and to better formulate entry and exit assertions* for each program component. If the design is not so defined, the debugging analyst must expend great effort to examine many modules at many levels to determine and formulate this information.

The methods used for representing the resulting design specifications vary. The specification may be English prose, a manual or automated program design notation, or graphical design representations, (e.g., flow charts, Nassi-Schneiderman diagrams, HIPO diagrams, structure diagrams) or some combination of these. The specification should be well documented because it will be of significant use to the debugging analyst in determining the correctness and consistency of the translation of design to source language specification. An additional set of applicable documents which may accompany design specifications in the military arena are the Interface Control Documents (ICDs) which contain descriptions of interconfiguration item interfaces.

* Entry assertions are those values of global data and input parameters which represent the assumptions inherent in the interface of one module with another. Exit assertions are those values of data set by the module in the implementation of a given function.

2.1.3 Source Language Specification/CPCI Test Procedures

The programmer translates the detailed design of an individual program module, as specified in the design specification, into a coded representation in the programming language associated with an abstract machine. (See Section 2.2 for a description of abstract machine.) The programming language used in most software development efforts is a higher order language (HOL) which incorporates language features which enable the programmer to reference a subroutine library and operating system functions, although in some developments, an assembly language is used instead of an HOL. This coded representation is referred to in this report as the source language specification. It includes the implementation of the design, as well as the needed instructions (e.g., job control language, data management, overlay directives) to the operating system. As such, the source language specification generally represents the final human transformation of the problem statement (requirements) into an executable implementation of the problem solution (design). The readability and reliability of this final representation is greatly related to the quality of the transformation process (i.e., requirements analysis and design) which preceded the coding phase. The source language specification is also influenced by the suitability of the programming language chosen for implementing the design. Even when the language of the design is very similar to the programming language, the transformation from a design for controlling data objects and data operations to an executable computer program is often as much as art as a science.

A technique currently in practice to bring older, but widely used, programming languages into closer correspondence to modern design and programming techniques is to use a structured programming language preprocessor. This type of tool overlays a language containing structured programming constructs on a language (e.g., FORTRAN) which may be deficient in structured language constructs so that it better supports structured programming. A programming language preprocessor may provide an enrichment to an existing programming language, thereby eliminating the need to change the language compiler to support structured programming. The preprocessor subsequently translates a program written using the structured programming constructs into a form consisting solely of the existing (e.g., FORTRAN) program language constructs. This technique may ease the programming task and even improve the software's reliability, but it can often make the job of debugging more difficult. A program written in a preprocessor language has essentially <u>two</u> source language representations, one written in the preprocessor language and the other written in the existing HOL, and the correspondence between them must be maintained. The debugging analyst must become familiar with both representations. In addition, many of the most effective debugging tools used by the debugging analyst present information on the state of the program's operation in terms of an HOL (see Section 2.2 and 2.3). If the debugging analyst is examining the preprocessor representation, the information these tools provide may not be immediately usable because an additional transformation on the data must be manually performed.

If one develops a software system in a top-down, hierarchical manner, the requirements for testing the software become more detailed in the same manner as the detail of the system design. This allows for an organized and comprehensive testing scheme to be developed. When the testing process is controlled, the debugging process also becomes more controlled. The CPCI Test Plans,

written earlier to correspond to the testing of acceptance criteria, are augmented by test procedures which specify how each test will be performed. The Plans and Procedures are the means for controlling the testing process. Generally, this in-depth planning of the testing process is documented in the CPCI Test Procedures which formulates specific preliminary and formal tests designed to demonstrate that the software meets the performance/functional requirements. It is the test procedures in combination with specific test plans which become the baseline against which integration testing is applied, and against which actual software behavior is compared with specified performance.

As referred to in this report, the CPCI Test Procedures are associated with the source language specifications because they are developed at approximately the same time and represent how the implementation of the design is to be verified. The CPCI Test Procedures should:

- Specify a test procedure for each performance requirement. (Generally, each procedure will cover one or more functional areas).

- Specify the acceptance criteria for each test case, and in those cases where the criteria are quantitative in nature, tolerances which define the range of acceptable performance must be provided. (These acceptance criteria should relate directly to the requirements specification.)

- Contain test cases designed to demonstrate the acceptability of the logic, computations, data handling, interfaces, and data base integrity for the CPCI.

- Specify the test data to be used.

- State the method for determining if each test condition is met, e.g., visual observation, data reduction and analysis, special timing analysis. This part of the test procedure should also be compatible with sections of the CPCI Development Specification.

- Identify the applicable references to the associated test plan, CPCI Development Specification, users manuals, positional handbooks, and other documentation for support programs or equipment.

These CPCI test procedures must be continually updated as ECP's are installed so they can retain their value throughout the period of integration-level testing and debugging. Additional documents (available in the military arena) which are included in this study with source language specifications are:

- <u>Version Description Document</u>. A description of the required hardware/software configuration for the software system, the new capabilities of an upgraded system, the open software problems, and other user-oriented information regarding a version of a software package.

- <u>Users Manuals</u>. A description of how the software is operated.

2.2 ABSTRACT MACHINE DESCRIPTIONS

A source program specification is in an implementation format which is machine processable. Machine processable implies that a hardware/software configuration and using procedures exist which will accept the source program specification and execute the operations it specifies. This report refers to the hardware/software configuration and using procedures which define the debugging analyst's computing environment as the abstract machine. Because the debugging methodology is generic in nature, and is intended to be applicable to diverse computing environments, a generic term is necessary to describe all hardware/software configurations that can be involved in debugging. A discussion of the impact that some hardware/software configurations (i.e., environments) have on the debugging methodology is presented in Section 4.

The abstract machine is the actual, physical hardware and software machine used by the programmer. The abstract machine may not be a single device such as a compiler, macro assembler or even a computer, but may be a combination of hardware and software (e.g., operating system and computer, compiler and subroutine (S/R) library). Possible abstract machines include the following:

- Micro-programmable computer
- Basic computer
- Basic computer and operating system
- Operating system and macro-library
- Compiler and operating system
- Compiler and S/R library
- Scientific support package

The programmer implements a problem solution representation corresponding to a specific abstract machine; optimally, the debugging analyst debugs a program in the same abstract machine used by the programmer. The lowest level physical machines are the micro-programmable machine and the basic computer; higher levels may be a higher order language with its associated compiler and a subroutine library. Associated with the physical machines are the sets of programmed instructions (i.e., software systems) which support the operation of the hardware. Those sets of programmed instructions vary according to hardware and functional capability properties and in their entirety, with the physical machine, constitute the abstract machine. The properties of an abstract machine, then, are reflected in the using instructions, (i.e., users manuals, associated with the hardware and sets of programmed instructions). As such, they comprise the abstract machine description, as shown in Figure 2-4.

The abstract machine most commonly used by the programmer is the higher order language (HOL) and its associated compiler. The programmer views this machine in terms of the capabilities provided by the programming language during detailed design and program code activities. The programmer may also use a

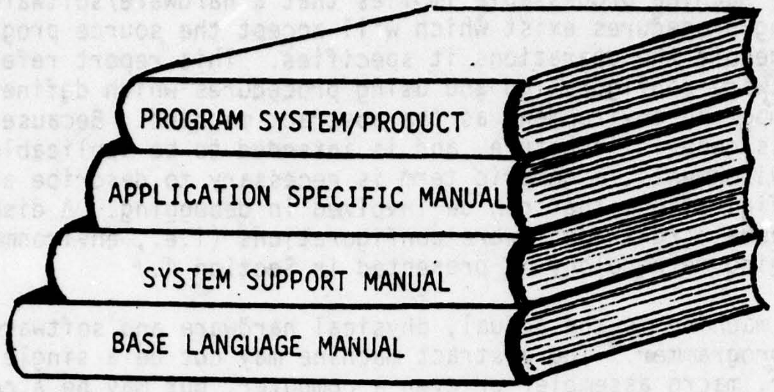


Figure 2-4. Abstract Machine Description

subroutine library, data base management services, etc., so that the abstract machine used is seen to be a complex hierarchy of functional capability in any one given environment. Its importance to debugging is that the abstract machine seen and used by the programmer (e.g., the HOL) may not be the same one seen and used by the debugging analyst (e.g., the basic computer and operating system), albeit, they may be the same person. The abstract machine used by the debugging analyst depends upon the functional capability of the software support system used in the software development process. In order for it to be identical to the one used by the programmer, it requires retention of data generated in the various forward translation processes from HOL to binary code (e.g., data dictionaries) to be later used to provide the inverse translations from binary code to HOL. This gives the debugging analyst the program execution information in terms of its source language representation. If support software does not exist to provide this capability, the debugging analyst is required to do the translation unassisted.

It can be seen that the use of software support tools can make the debugging process more efficient and reliable. For example, a data dictionary generated during the compilation process can be saved for a trace program which is called to collect and display the application program's name space values at particular points in execution. It is far easier and more reliable for the trace program to automatically convert the machine representation of the name space to the same HOL representation of symbolic names, data types, and values than for the analyst to do it manually, given that this is a much used debugging activity. As seen in Figure 2-5, the hierarchy of abstract machines that may exist in any given software development may be extremely complex, including, perhaps, interfaces with several external hardware systems (e.g., the ARPANET). The programmer views the hierarchy of abstract machines mainly

from the top levels and may not realize their complex relationship at lower levels, but the debugging analyst may need to understand the operation of many levels of the hierarchy in order to find the bug, or duplicate an error.

Since the hierarchy of abstract machines is sometimes hidden by the HOL used by the programmer, but is not to the debugging analyst, the analyst may need access to more abstract machine descriptions than the programmer. The transparency of the other abstract machines is a function of the HOL used by the programmer and the technique used by the compiler in the forward translation process. If the reverse translation process does not employ the same techniques, the abstract machines being used in debugging should be known and described in a comprehensive, up-to-date manner.

The complexity of the abstract machine impacts its own reliability in performance of functional capability. The debugging methodology presented in this report applies to integration phase testing of application software; however, it is sometimes common for the analyst to encounter errors in the implementation of an abstract machine functional capability rather than the application software. This is especially true when using newly developed abstract machine capabilities (i.e., compilers or operating systems). Duplication of application software performance often requires exact timings from both the application software and abstract machine. The more complex and interrelated the abstract machine is, the more difficult it is to duplicate exact computing incidences.

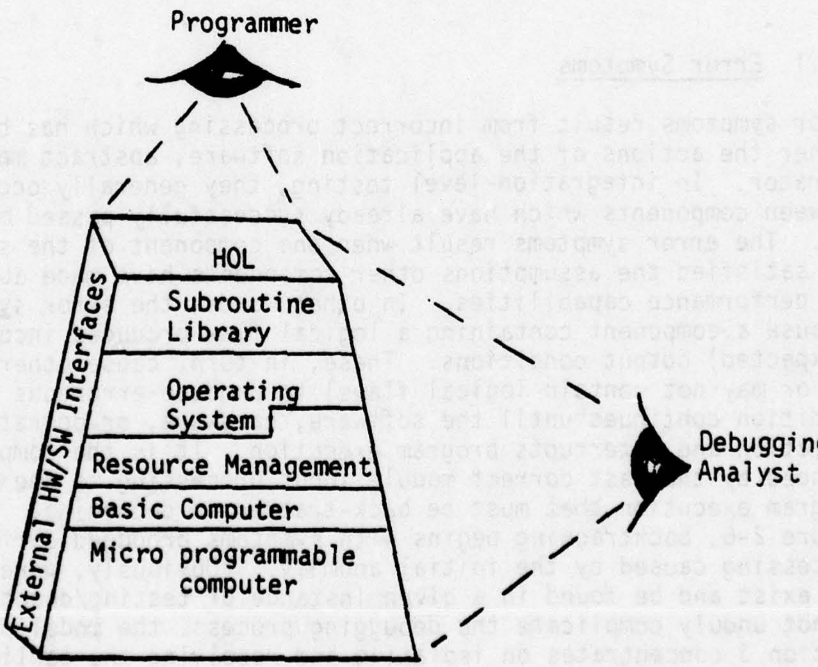


Figure 2-5. Hierarchy of Abstract Machines as Seen by the Programmer/Debugging Analyst

2.3 RUN-TIME INFORMATION

Run-time information, as referenced to in this report, is dynamic program information directly relating to a specific software problem. It consists of two components:

- Error Symptoms
- Program State Information

Run-time information is used throughout the debugging process. It initiates a specific instance of debugging by the issuance of a software discrepancy report which indicates incorrect software performance information (outputs) or error symptoms reflecting incorrect processing. Additional run-time information is generated during each instance of debugging in order to support derivation of other necessary debugging information components needed to isolate the anomalous software error. Final run-time information is produced to test the hypothesis (error cause) and to demonstrate the resolution of the software error.

Error symptoms are obtained by examining the external performance during the software's operation. Program state information is obtained by using special tools which capture and display internal operation of the software.

2.3.1 Error Symptoms

Error symptoms result from incorrect processing which has been detected by either the actions of the application software, abstract machine or computer operator. In integration-level testing, they generally occur at the interface between components which have already successfully passed module-level testing. The error symptoms result when one component of the system hierarchy has not satisfied the assumptions other components have made about its functional and performance capabilities. In other words, the error symptoms result because a component containing a logical flaw produces inconsistent (or unexpected) output conditions. These, in turn, cause other components (which may or may not contain logical flaws) to produce erroneous results. This condition continues until the software, hardware, or operator detects abnormal operation and interrupts program execution. It is the computational history bounded by the last correct module input processing to the interruption of the program execution that must be back-tracked in debugging. As shown in the Figure 2-6, backtracking begins with symptoms produced during the erroneous processing caused by the initial anomaly. (Obviously, more than one error may exist and be found in a given instance of testing/debugging. In order to not unduly complicate the debugging process, the model presented in Section 3 concentrates on isolating and resolving the earliest-occurring anomalous condition.)

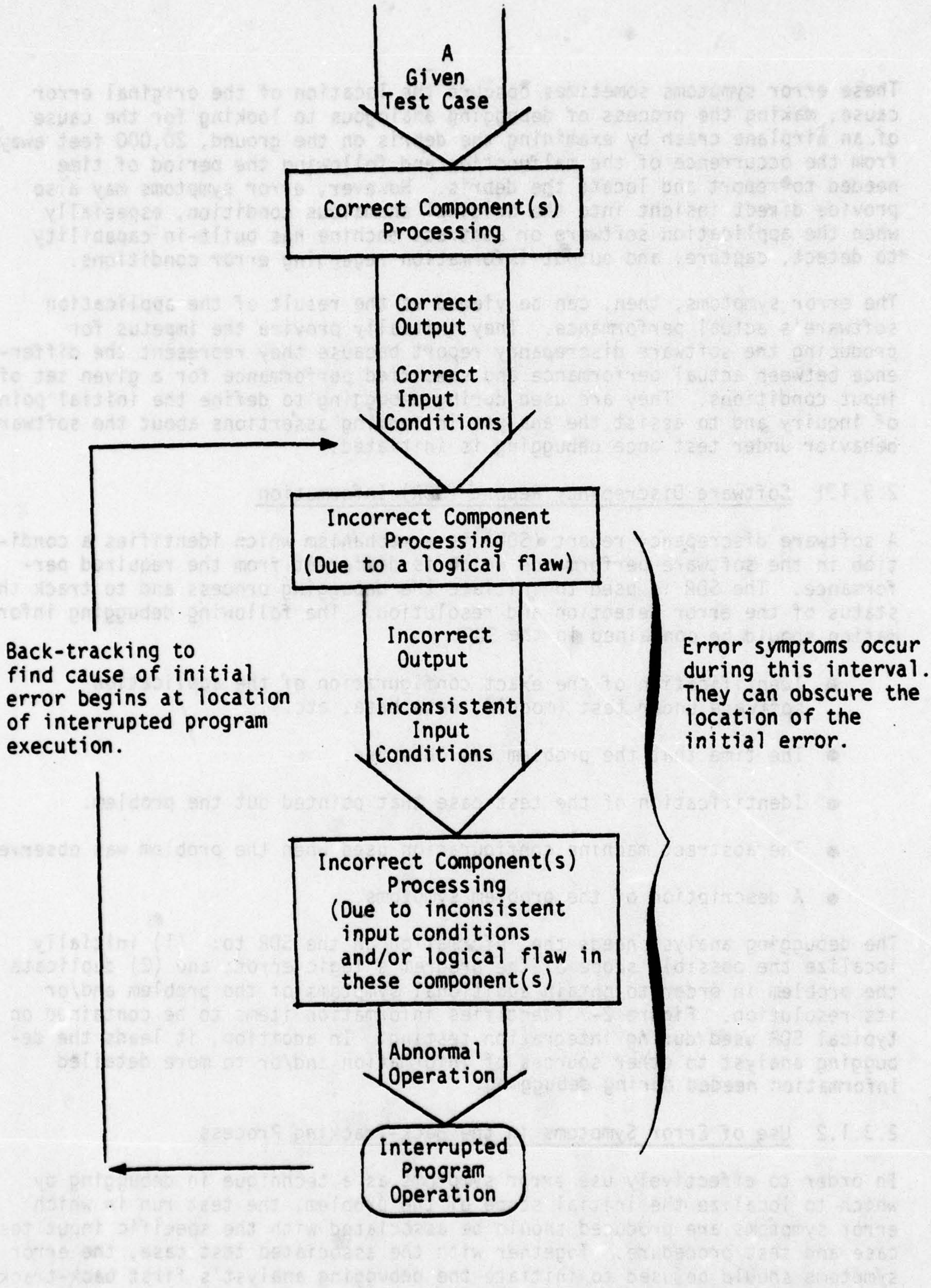


Figure 2-6. Backtracking Error Symptoms

These error symptoms sometimes obscure the location of the original error cause, making the process of debugging analogous to looking for the cause of an airplane crash by examining the debris on the ground, 20,000 feet away from the occurrence of the malfunction and following the period of time needed to report and locate the debris. However, error symptoms may also provide direct insight into the original anomalous condition, especially when the application software or abstract machine has built-in capability to detect, capture, and output information regarding error conditions.

The error symptoms, then, can be viewed as the result of the application software's actual performance. They initially provide the impetus for producing the software discrepancy report because they represent the difference between actual performance and specified performance for a given set of input conditions. They are used during debugging to define the initial point of inquiry and to assist the analyst in forming assertions about the software behavior under test once debugging is initiated.

2.3.1.1 Software Discrepancy Report (SDR) Information

A software discrepancy report (SDR) is a mechanism which identifies a condition in the software performance which is different from the required performance. The SDR is used to initiate the debugging process and to track the status of the error detection and resolution. The following debugging information should be contained in the SDR:

- Identification of the exact configuration of the application software under test (module, data base, etc.).
- The time that the problem was observed.
- Identification of the test case that pointed out the problem.
- The abstract machine configuration used when the problem was observed.
- A description of the problem symptoms.

The debugging analyst needs the information on the SDR to: (1) initially localize the possible scope of the program's logic error; and (2) duplicate the problem in order to obtain additional symptoms of the problem and/or its resolution. Figure 2-7 identifies information items to be contained on a typical SDR used during integration testing. In addition, it leads the debugging analyst to other sources of information and/or to more detailed information needed during debugging.

2.3.1.2 Use of Error Symptoms in the Back-Tracking Process

In order to effectively use error symptoms as a technique in debugging by which to localize the initial scope of the problem, the test run in which error symptoms are produced should be associated with the specific input test case and test procedure. Together with the associated test case, the error symptoms should be used to initiate the debugging analyst's first back-tracking activities. Association of test cases and test procedures with the

SDR Number - A unique number assigned by Configuration Management (CM).

Date - The date the SDR is logged by CM.

Time - Time of day the problem was discovered.

Originator - Name of the originator of the SDR.

Status - Used to track the status of the SDR such as open, closed, deferred.

Test Phase* - Test phase during which the problem was discovered, including: integration testing, system testing, validation testing, acceptance testing.

Problem ID* - Identification or location of the problem, including module, data base, document, or some combination of these.

Element in Error* - Name of the element exhibiting the problem.

(If the specific module, data base, document is not known, identify subsystem).

Priority - Indicates priority for fix, including low, medium, or high.

Mod* - Modification of module, data base, etc., exhibiting the problem, if known.

Test Configuration* - ID of the test case(s) and test procedure step(s) which demonstrated the problem.

Software Configuration* - ID of software versions which exhibited the problem.

Hardware Configuration* - ID of computer systems being operated when the problem was discovered.

Problem Description - In terms of the primary symptoms, apparent secondary symptoms and, if possible, a listing or description of the actual problem.

Other Open SDRs - References to other SDRs which have not been resolved and which are directly or indirectly associated with the contents of this SDR.

* Establishes test environment.

Figure 2-7. Typical Information on an SDR

requirements, perhaps by way of traceability matrices, provides the mechanism by which the error symptoms can be related to a specific functional component, thereby defining the initial point of inquiry. Since isolation of the faulty software component is one aspect of integration-level debugging, optimizing the analyst's efficiency in isolating that component is one objective of the debugging methodology.

A useful adjunct to a debugging methodology for integration-level testing is a mechanism for correlating error symptoms with error causes. In addition, one would assume that integration-level program errors and associated error symptoms should be significantly different than those found in module-level testing/debugging. For example, one would generally expect that the infinite program loop, which is common in module-level testing, would appear on a less frequent basis than other error symptoms. However, this study could find no references for correlating integration-level error causes to error symptoms. The wide scope of errors which this report suggests does not provide a detailed mechanism by which to use the error symptoms during debugging. (This would require an in-depth study of error causes/symptoms for specific types of applications during integration testing). Further, the relationship between tools and techniques useful for debugging during integration-testing and error symptoms/causes were not defined in the literature search. Consequently, no correlation has been made in this study between error symptoms and debugging tool usage.

While this study could not find a correlation between error symptoms, error causes and debugging tool usage, it does define and classify the types of errors found in debugging. An error found in integration-level testing and debugging is generally defined as a failure to satisfy a functional/performance specification. It may be a result of:

- An implementation error in the program specification.
- Failure to write specifications that correctly represent a design.
- Failure to conform to tolerance limits as represented by the design specifications.
- Misinterpretation and/or inaccurate expression of the functional/performance requirement as represented by the requirements specification.

An implementation error is a broad category of error concerning the source program representation of interfacing functional components in a given environment to a set of input conditions. It can consist of errors in the logic, interfaces, data definitions, and/or control structure assumptions one component makes of another component. These errors are to be expected in integration-level listing. They are usually isolated and resolved in a relatively short period of time. An implementation error generally impacts only the source program specification.

Errors caused from failure to write specifications that correctly represent a design may represent both design ambiguities, omissions and inconsistencies. Often, this type of error results when the designer is different than the

implementer. An example is insufficient error processing of the input conditions, resulting in processing of input data which should have been rejected. Another example is numerical scaling problems where the degree of mathematical precision required is not obtained because the implementer's understanding of the abstract machine's mathematical processing is less sophisticated than the designer's, who omitted detailed mathematical manipulation in the design specification. This type of error has diverse symptoms, is generally more difficult and costly to correct than the preceding implementation error type, and impacts the source program specification and possibly the design specification.

Errors caused from non-conformance to tolerance constraints represent a serious design flaw. These types of errors generally result from inadequate trade-off and feasibility studies in the early requirements analysis process. They are usually very costly to correct as they almost always require redesign, and may even require changes to the hardware configuration. A characteristic of testing for tolerance constraints is that often the software has to be coded and executed before tolerance deficiencies are found. That is, design walk-throughs and inspection cannot indicate whether performance tolerance requirements are satisfied. Debugging of errors associated with performance tolerance requirements are accomplished in the same manner as debugging other types of errors; resolution of the error cause, however, often requires reanalysis, redesign, recode, and retest, with extensive changes to the design and source program specifications.

Errors caused from the designer's failure to correctly interpret a functional performance requirement causes symptoms directly related to the functional/performance requirement. It impacts the design specification and the source program specification. The error symptoms may consist of a small deviation in output to missing or incorrect output. The debugging effort associated with this type of error is generally more difficult and expensive as its resolution requires a design change.

As previously stated, this study could not find existing studies which demonstrated any correlation between the error causes and resulting error symptoms. The error causes, however, do indicate why integration-level errors are more difficult and expensive to isolate and resolve than are module-level errors. The isolation and resolution of errors found in integration-testing may involve many program components and several levels of system specifications; and the isolation of the error cause may be equally as difficult an activity as those associated with resolution of the error. Whereas module-level testing implies the application of a series of tests designed to exercise a range of a module's input values, all logical decisions, and control paths of a single functional component, integration-level testing implies testing the combination of functional components. Module-level testing is performed by a single programmer, usually in a well defined manner (e.g., driver and program module as in bottom-up development), and errors found are limited to the single program component, its design specification, and/or its related requirements specification. Integration-level testing is performed, usually, by many people, and involves the entire software system, and all related documentation.

2.3.2 Program State Information

Program state information is defined as the values of the name space (data variables) of a program at any specific point (in time or operational sequence) in that program's execution. It represents a static picture of an implementation of an abstract software component (i.e., functional capability) at a given point in response to a given set of input conditions. Program state information is captured during execution of program statements. A collection of program state information recorded at several points in a program's sequential operation represents a sequence of operations and the resulting consequence of those operations as values in the program's name space. Program state information is a key element in the debugging process because it provides information on actual program performance by way of values generated for program variables as a consequence of processing given inputs in such a manner so as to obtain specified outputs. It is essential to the debugging process because it provides information on the intermediate operations performed and results (i.e., values of data variables) generated by the specific software implementation under test. Program state information can be obtained by debugging support software, such as hardware or software data collection monitors.

Program state information is best represented in a format which allows the debugging analyst to understand the software in terms of its software component abstraction and implementation. That is, if the abstract software component is implemented in an higher order language (HOL), the program state information representing the processing required to implement the function should be returned to the analyst in the same *HOL representation*. This requires the debugging software use a translation process related to the one which enabled the programmer to implement a design in a language which is not binary in format. (See Section 2.2, Abstract Machine Description).

If programs could be written to contain no errors, debugging would never occur and the need to examine program states at intermediate points in a given test execution would not be necessary. The final results (i.e., the values for the final output variables) are sufficient evidence to determine program correctness. In fact, upon occasion this has occurred. However, programs are not generally written correctly and it is common that in order to determine the condition that produced incorrect results, the test execution must be repeated, and/or sometimes altered, by the debugging analyst in order to obtain program state information at given points in time, both prior to and following the occurrence of a software error. Isolation of the error requires the analyst to identify the software component which incorrectly implements an abstraction; detection of the error requires the analyst to resolve inconsistencies in the implementation of the abstraction with regards to time, place, and space dimensions. In other words, the analyst must reconstruct the computational history of the software to determine the time, place, and condition responsible for the anomaly. In so doing, he must often generate evidence reflecting the values of a subset of the program's name space to a set of known conditions at specific point(s) in the software's execution.

The following section discusses the types of program state information generally required for the isolation and detection of a software error.

2.3.2.1 Elements of Program State Information

Program state information consists of the following elements:

- Symptom Locality. The point in real time* and the location in real space* to which all other program state information relates, i.e., the termination point in a specific traversed path at a given time to given set of input conditions.

- Traversed Paths. Information about the control flow, or program paths traversed, in response to a given set of input conditions.

- Name-Space Values. Values of specific variables set in the traversal of a logical path in a program component in response to a given set of input conditions. It represents the specific implementation of a design of the problem solution on a specific abstract machine. Of particular interest to debugging are the interfaces between the structures provided by one level of the abstract machine to set and use name space values in the application software and the structures provided by a lower level.

- External System Status. Information pertaining to the status of external hardware/software systems at the symptom locality.

- Simultaneous Events. Information relating to combinations of the above factors when simultaneous software executions are involved.

- Space Allocation and Timing. Information regarding the allocation of software components to the hardware devices and the execution timing of the software at symptom locality.

---

* The terms "real time" and "real space" are used to highlight the requirement in debugging to determine the actual point in the program's operation used as a reference for program state information. The programmer does not have the requirement to use actual time and space. The program can be written in terms of abstract or virtual time and space.

2.3.2.2 Program State Representations

Program state data at the machine level is represented by binary code. That format may be least usable to the debugging analyst in terms of representing a specific implementation of a functional capability. That is, the debugging analyst needs first to look for an error in the specific implementation of a design (i.e., the source language specification). Program state information should be presented to him in terms of that specification, i.e., the source language program. The source language program should attempt to use terms commonly defined and understood within the context of its functionality in relation to the system and its description. That is, an observable connection between source program listing and design description must necessarily exist because the error source may not be in the implementation but in the translation of the design specification to the source language representation.

Once the debugging analyst confirms the correctness of the source language implementation, on an abstract machine, he must examine the translation of the design specification to source language implementation for correctness and completeness. If no error is found in that translation, he must then examine the translation from requirements specification to design specification. The debugging analyst cannot use program state information to verify these transformations since the information cannot be represented in a format that corresponds to either the design or requirements specifications.

The software development process consists of a series of problem solution specifications, originating with the problem statement (i.e., the required operational capability) proceeding to the lowest level of abstract machine representation - binary code, and possibly micro-code. During the integration testing and debugging process, the analyst usually needs to examine the problem statement in at least three representations, that is requirements, design, and source program language specifications. At the current state-of-the-art in language specification and processing, there is a major division between those representations that are processable only by a human and those which are processable by a computer, as shown in the Figure 2-8. The division is distinguished by the degree that there exists an automated capability to translate a higher level specification of the abstract problem solution to a lower level specification. For example, programs exist that translate HOL source statements to machine instructions. Programs also exist that can translate program state information at the machine level to a HOL-level, providing that the reverse translation was made first. However, at the present time, there are no tools for automatically translating a design specification to HOL source statements. Consequently, program state information cannot be represented at the design specification level which requires the debugging analyst to be responsible for detecting logical flaws in (usually) another analyst's design of the problem solution without the direct aid of automated tools. (It should be noted that there is much research activity in the techniques of automatically translating from a design specification to HOL source statements and in the future, computerized translation of representations may be moved to a higher level of abstraction). (See Section 2.1 for a discussion of techniques that can be used to verify the correctness of both translations as well as the internal consistency of each specification).

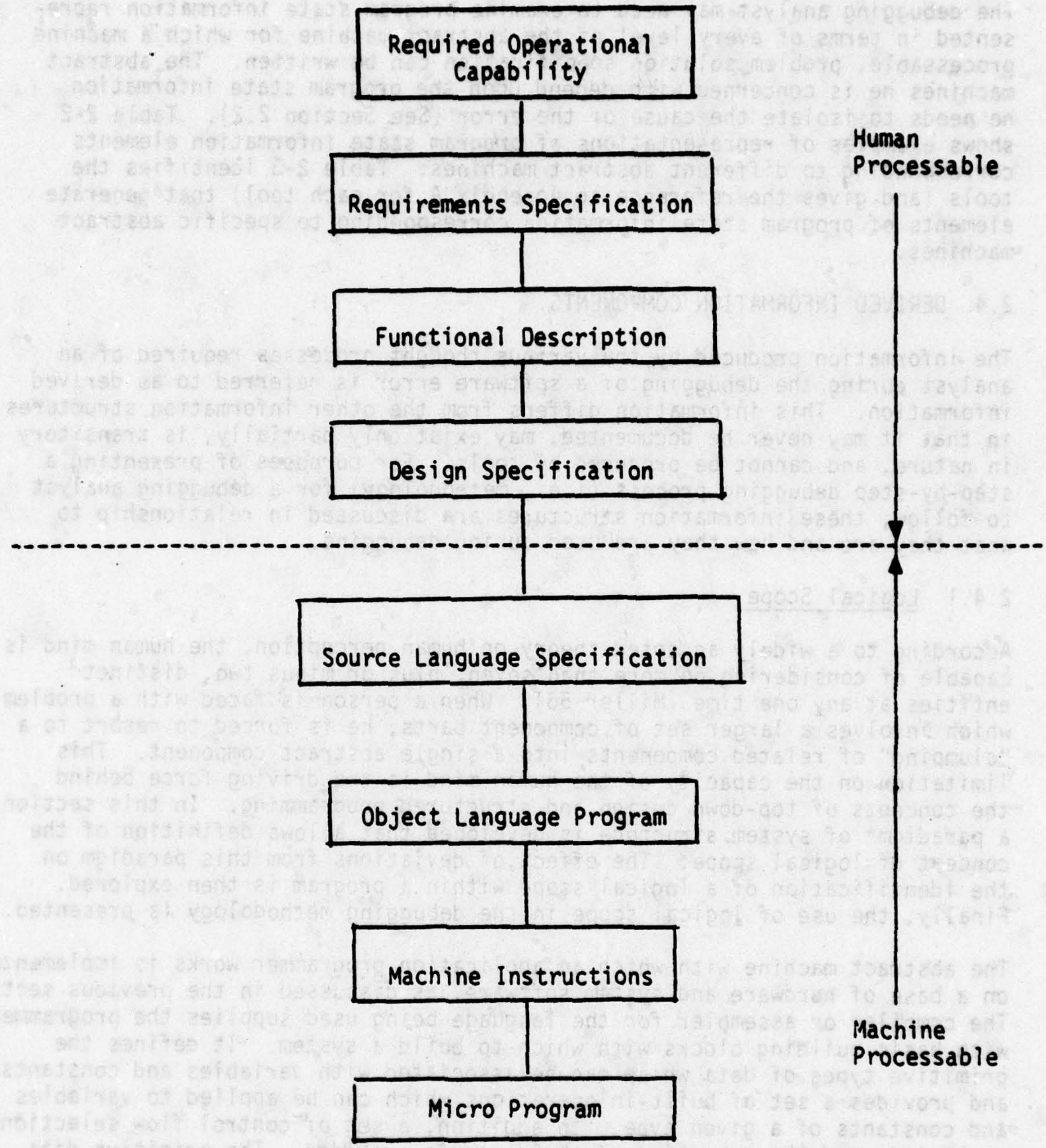


Figure 2-8. Hierarchy of Problem Solution Statements

The debugging analyst may need to examine program state information represented in terms of every level of the abstract machine for which a machine processable, problem solution specification can be written. The abstract machines he is concerned with depend upon the program state information he needs to isolate the cause of the error (See Section 2.2). Table 2-2 shows examples of representations of program state information elements corresponding to different abstract machines. Table 2-3 identifies the tools (and gives the reference to Appendix A for each tool) that generate elements of program state information corresponding to specific abstract machines.

2.4 DERIVED INFORMATION COMPONENTS

The information produced by the various thought processes required of an analyst during the debugging of a software error is referred to as derived information. This information differs from the other information structures in that it may never be documented, may exist only partially, is transitory in nature, and cannot be produced by tools. For purposes of presenting a step-by-step debugging process (i.e., methodology) for a debugging analyst to follow, these information structures are discussed in relationship to what they are and how they are used during debugging.

2.4.1 Logical Scope

According to a widely accepted theory on human perception, the human mind is capable of considering no more than seven, plus or minus two, distinct entities at any one time (Miller 56). When a person is faced with a problem which involves a larger set of component parts, he is forced to resort to a "clumping" of related components into a single abstract component. This limitation on the capacity of the human mind is the driving force behind the concepts of top-down design and structured programming. In this section, a paradigm* of system structure is developed that allows definition of the concept of logical scope. The effect of deviations from this paradigm on the identification of a logical scope within a program is then explored. Finally, the use of logical scope in the debugging methodology is presented.

The abstract machine with which an application programmer works is implemented on a base of hardware and system software, as discussed in the previous section. The compiler or assembler for the language being used supplies the programmer with basic building blocks with which to build a system. It defines the primitive types of data which can be associated with variables and constants, and provides a set of built-in operations which can be applied to variables and constants of a given type. In addition, a set of control flow selection mechanisms and the semantics of their use is provided. The primitive data types provided by a compiler must be assembled into other more abstract structures which represent the types of data which are operated upon by the

* paradigm - a typical structural pattern.

TABLE 2-2. Examples of Program State Representations

Abstract Machine	Program State Information Elements					
	Traversed Paths	Symptom Locality	Name-Space Values	External System Status	Simultaneous Events	Space Allocation and Timing
Micro Programmable Computer	Micro-instruction addresses	Micro-inst. address	Flip-flop values Data Bus registers ROM contents RAM contents	Interface register contents	Internal register usage	Control memory maps
Basic Computer	Location counters (memory address)	Memory address	Machine registers Memory contents Secondary storage contents	Interrupts I/O buffer Status registers	Memory location references Secondary storage references	Physical memory maps Instruction timing
Operating System	*Schedule queues* Program operation messages	Resident program	I/O transfer buffers	Enabled/disabled indicator	Shared name-space of multiple programs	Program execution timing data
Assembler	Instruction labels	Instruction address	Assembly labels and types	Buffer status words	Test & Set instruction	Memory map
Macro Library	Macro names	Macro name	Macro parameter values	Error messages	Semaphores	Memory map
Compiler	Statement labels Procedures names Block ID	Procedure names Statement label	HOL data labels and type	Built-in on condition	On Conditions Multi-tasking primitives	Memory map
Special Support Packages	Error messages	Error message	Data structure and values	Error message	Message buffer contents	Memory map

Table 2-3. Applicable Debugging System Components

	Program State Information					
Abstract Machine	Traversed Paths	Symptom Locality	Name-Space Values	External System Status	Simultaneous Events	Space Allocation and Timing
Micro Programmable Computer	Computer Emulators A.13 Computer Simulators A.14		Computer Emulators A.13 Computer Simulators A.14 Monitor Dump A.5 H/W Monitor A.11	Computer Emulators A.13 Computer Simulators A.14 Monitor Dump A.5 H/W Monitor - A.11	Monitor dump A.5 H/W Monitors A.11	Hardware monitor A.11(I)
Basic Computer	Dynamic Internal traces A.7,8,9, 18,22 Monitor computer A.12 Computer Emulator A.13 Computer Simulator A.14 Software Breakpoint/Traps A.16 Interactive Modification A.19, A.20,21	Post-mortem dumps A.1 Trap/Breakpoint Dumps A.3 S/W Breakpoint traps A.16 Comparator Tools A.25	Post-mortem Dumps A.1 Snap Shot Dumps A.2 Trap/Breakpoint Dumps A.3 Monitor Dumps A.5 Auxiliary Storage/ Utility Dump A.6 Dynamic Internal Traces A.7,8,9, 18,22 H/W Monitors A.11 Monitor Computer A.12 Computer Emulator A.13 Computer Simulator A.14 S/W Breakpoint/ Traps A.16 Interactive Modification A.19, 20,21	Post-mortem dumps A.1 Snap Shot Dumps A.2 Trap/Breakpoint Dumps A.3 Monitor Dumps A.5 Auxiliary Storage/ Utility Dumps A.6 Dynamic Internal Traces A.7,8,9, 18,22 H/W Monitors A.11 Monitor Computer A.12 Computer Emulator A.13 Computer Simulator A.14 S/W Breakpoint/Traps A.16 Interactive Modification A.19,20 21	Snap Shot Dumps A.2 Trap/Breakpoint Dumps A.3 Monitor Dumps A.5 H/W Monitors A.11 S/W Breakpoint/Traps A.16	Post-mortem Dumps A.1(S) Trap/Breakpoint Dumps A.3(S) H/W Monitors A.11 (T) Monitor Computer A.12(T) Comparator Tools A.25(S)
Operating System	Programmed-In Dumps A.4 Dynamic Internal Traces A.7,8,9, 18,22 S/W Breakpoint/ Traps A.16 Interactive Modification A.19,20,21	Post-mortem Dumps A.1 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 Set-Use Matrix/ Cross Reference Analysis A.10 S/W Breakpoint/ Traps A.16 Comparator Tools A.25	Post-mortem Dumps A.1 Snap Shot Dumps A.2 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 Auxiliary Storage/ Utility Dump A.6 Dynamic Internal Traces A.7,8,9,18, 22 Set-Use Matrix/Cross Reference Analysis A.10 S/W Breakpoint/Traps A.16 Interactive Mod. A.19, 20,21	Post-mortem Dump A.1 Snap Shot Dump A.2 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 Auxiliary Storage/ Utility Dump A.6 Dynamic Internal Traces A.7,8,9,18,22 S/W Breakpoint Traps A.16 Interactive Mod. A.19 20,21	Snap Shot Dumps A.2 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 S/W Breakpoint/Traps A.16	Post-mortem Dumps A.1(S) Trap/Breakpoint Dump A.3(S) Programmed-In Dumps A.4(S,T) Comparator Tools A.25(S)

Table 2-3. Applicable Debugging System Components (Cont'd)

	Program State Information					
Abstract Machine	Traversed Paths	Symptom Locality	Name-Space Values	External System Status	Simultaneous Events	Space Allocation and Timing
Assembler	Programmed-In Dumps A.4 Dynamic Internal Traces A.7,8,9, 18,22 Monitor Computer A.12 S/W Breakpoint/ Traps A.16 Interactive Modification A.19 A.20,21	Post-mortem Dumps A.1 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 Set-Use Matrix/ Cross Reference Analysis A.10 S/W Breakpoint/ Traps A.16 Comparator Tools A.25	Post-mortem Dumps A.1 Snap-Shot Dumps A.2 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 Dynamic Internal Traces A.7,8,9,18, 22 Set-Use Matrix/ Cross Reference Analysis A.10 Monitor Computer A.12 S/W Breakpoint/ Traps A.16 Interactive Mod. A.19,20,21	Post-mortem Dumps A.1 Snap-Shot Dumps A.2 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 Dynamic Internal Traces A.7,8,9,18, 22 Monitor Computer A.12 S/W Breakpoint Traps A.16 Interactive Mod. A.19,20,21	Snap-Shot Dumps A.2 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 S/W Breakpoint/Traps A.16	Post-mortem Dumps A.1(S) Trap/Breakpoint Dumps A.3(S) Programmed-In Dumps A.4(S,T) Monitor Computer A.12(T) Comparator Tools A.25(S)
Macro Library/ Subroutines	Programmed-In Dumps A.4 Monitor Computer A.12 S/W Breakpoint/ Traps A.16 Interactive Modification A.19,20,21	Post-mortem Dumps A.1 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 Set-Use Matrix/ Cross Reference Analysis A.10 S/W Breakpoint/ Traps A.16	Post-mortem Dumps A.1 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 Auxiliary Storage/ Utility Dump A.6 Dynamic Internal Traces A.7,8,9,18, 22 Set-Use Matrix/ Cross Reference Analysis A.10 Monitor Computer A.12 S/W Breakpoint/ Traps A.16 Interactive Mod. A.19,20,21	Post-mortem Dumps A.1 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 Auxiliary Storage/ Utility Dump A.6 Dynamic Internal Traces A.7,8,9,18,22 Monitor Computer A.12 S/W Breakpoint Traps A.16 Interactive Mod. A.19 20,21	S/W Breakpoint/Traps A.16	Post-mortem Dumps A.1(S) Trap/Breakpoint Dump A.3(S) Programmed-In Dumps A.4(S,T) Monitor Computer A.12(T)

Table 2-3. Applicable Debugging System Components (Cont'd)

Abstract Machine	Program State Information					
	Traversed Paths	Symptom Locality	Name-Space Values	External System Status	Simultaneous Events	Space Allocation and Timing
Compiler/ Interpreter	Programmed-In Dumps A.4 Monitor Computer A.12 S/W Breakpoint/ Traps A.16 Interactive Modification A.19, A.20,21	Post-mortem Dumps A.1 Trap/Breakpoint Dump A.3 Programmed-In Dumps A.4 Set-Use Matrix/ Cross Reference Analysis A.10 S/W Breakpoint/ Traps A.16 Comparator Tools A.25	Post-mortem Dumps A.1 Snap Shot Dumps A.2 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 Auxiliary Storage/ Utility Dump A.6 Dynamic Internal Traces A.7,8,9,18, 22 Set-Use Matrix/ Cross Reference Analysis A.10 Monitor Computer A.12 S/W Breakpoint/ Traps A.16 Interactive Modification A.19, 20,21	Post-mortem Dumps A.1 Snap Shot Dumps A.2 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 Auxiliary Storage/ Utility Dump A.6 Dynamic Internal Traces A.7,8,9,18, 22 Monitor Computer A.12 S/W Breakpoint/ Traps A.16 Interactive Mod. A.19,20,21	Snap Shot Dumps A.2 Trap/Breakpoint Dumps A.3 Programmed-In Dumps A.4 S/W Breakpoint/Traps A.16	Post-mortem Dumps A.1(S) Trap/Breakpoint Dumps A.3(S) Programmed-In Dumps A.4(S,T) Monitor Computer A.12(T) Comparator Tools A.25(S)

application program. Application-level operations and control mechanisms can then be defined by procedures of the application program.

Data structures and procedures can be viewed as implementing components or modules of a larger system. Such components may themselves be composed of smaller modules, or may be used in the creation of larger modules. Each such component can be endowed with an abstract meaning, i.e., a single concept representing the functional role of the component. In a well structured program, any given functional element of the program can be viewed as a small number of these components which interact in limited, well defined ways. The concept of logical scope, as used in the debugging methodology study, is the identification of such a functional area of a program and/or its subcomponents to be studied for correctness.

The progression beginning with the data types and operations provided by the compiler and proceeding towards the data types and operations suited to the application can be viewed as the development by a programmer of a hierarchy of increasingly more abstract data types, each using the types of its lower level as building blocks. If the system is well structured, then each component consists of a small number of variables of various abstract data types and a control structure which defines the manner in which they are used. Ideally, each such component is small enough to be understood and the process of debugging need only involve the consideration of one such component at a time. If its abstract subcomponents interact in a well-defined way, then an error in a component should be one of three types:

- An inappropriate algorithm for the application in terms of data type and operations used.
- Incorrect control flow and control dependency or data flow and data dependency.
- Use of a subcomponent which fails to support the abstraction associated with it.

In the first two cases, the error lies in the component being studied. In the third case, the problem lies lower in the hierarchy of the program, and the analyst must turn his attention to the control structure, algorithm, and data types of the offending subcomponent.

Within this paradigm, the concept of logical scope is simply the identification of the component to be analyzed. This paradigm of program structure implies several very strong assumptions with regard to the methodology presented in Section 3. These include:

- Program components interact only in well-defined ways.
- The operation of one component does not depend upon the internal structure of another component for its correct behavior. This is equivalent to saying that components are defined by an explicit input/output relation, and can be replaced by any component satisfying this relation.

- The operation of a component depends upon correct establishment of external values by interfacing components.

In order for the concept of logical scope to be useful in debugging, the behavior of a functional program capability must be understandable in terms of the abstract meaning assigned to its components. Further, the number of these components must be small enough to be comprehended by a human analyst (i.e., around seven plus or minus two). Violation of any of the assumptions mentioned earlier can make this goal unreachable. If program components interact in ways that go beyond the abstract meaning assigned to them, then it is necessary to consider the functional capability in terms of the behavior of the subcomponents of its own components. The number of components to be analyzed can then grow geometrically often exceeding the capacity of the analyst to fully understand their mutual behavior. If the operation of one component depends upon the internal structure of another, then changes made to remedy a bug in one component may introduce new errors in other components. As an example of the impact of structure on logical scope, consider Figure 2-9. Figure 2-9(a) shows a complicated system consisting of 5 components, each of which depends directly on the behavior of all other components for its correct operation. For the sake of discussion, assume that an average of 10 simple steps of reasoning or testing are required to understand the relationship between one component and another and to verify that they are properly connected. Then the verification of component C1 requires 40 steps and the verification of the system requires 200 steps of reasoning. By way of contrast the verification of the system portrayed in Figure 2-9(b) requires only 20 steps of reasoning for component C1 and 40 steps to verify the entire system.

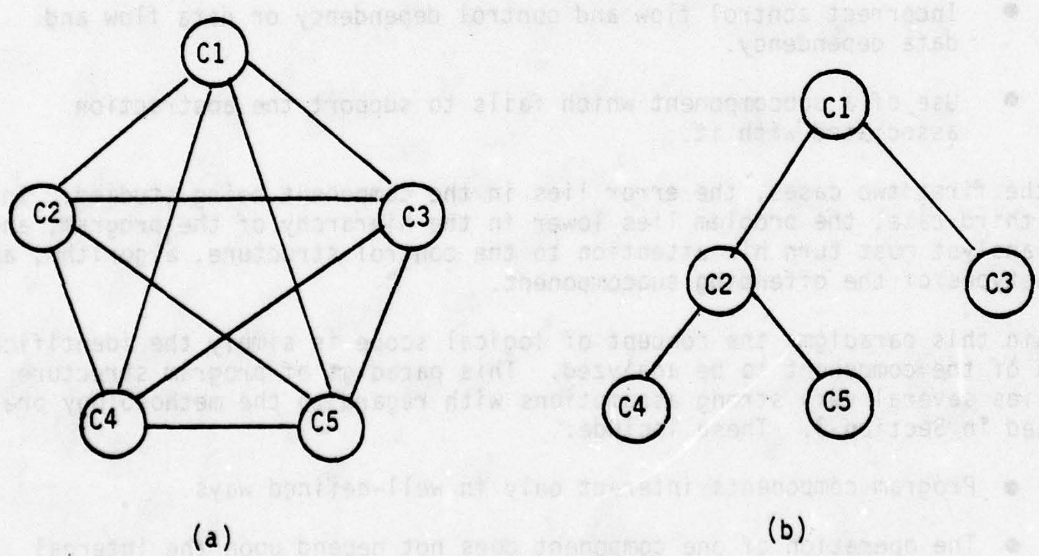


Figure 2-9. Impact of Structure on Logic Scope

With relationship to the debugging methodology, the logical scope defines the functional component, or subcomponent, or segment thereof, which will be analyzed to a greater depth with the intent of isolating and, then, resolving an observed software error. The logical scope is a mechnaism by which to logically define the segment of source program specification, and related design and requirements specifications, on which the debugging analyst will focus in his investigation of the computational history, in reverse, to find the (earliest occurring) software error. Logical scope, then, defines a progression of functional components which must be examined during back-tracking to find the bug.

In summary, the concept of logical scope means the identification of a functional area of a program and all of its subcomponents that must be understood in order to understand the behavior of a functional area. The degree to which the behavior of the functional area can be comprehended by a human analyst is critically dependent upon the simplicity and clarity of structure of the program. It is used during back-tracking to define the current focus of investigation. It is a product of the human mind and may result as a flash of quick insight or by detailed study of other information components.

2.4.2 Assertions

In the process of debugging a problem within a defined logical scope, it is necessary to compare program state information, as collected by debugging tools, with the analyst's expectations of correct program states at a given point of program execution. These expectations take the form of assertions about the values that program variables (real space) *should have at a given point* (real time) in the program in the presence of the input data defined by the current test case. There are three kinds of assertions that are required of the analyst according to the methodology presented in the debugging model. They include:

- An entry assertion is a statement of the assumptions a component makes about the state of its environment when it is invoked.

- An exit assertion is a statement of the effect a component has had on its environment when it terminates.

- An intermediate assertion is an entry or exit assertion about the behavior of a components' subcomponents. The impact of intermediate assertions should be that they collectively imply that the truth of the component's entry assertion assures the truth of the interfacing component's exit assertion, or that the component's exit assertions assures the truth of all subcomponents' exit assertions.

The assertions generated during debugging are associated with the debugging analyst's investigation of the performance of the functional component defined by the logical scope (i.e., a bounded segment of program code). They represent his understanding of the implementation in terms of its computing environment according to a design in response to specified functional/performance requirements. The assertions take the form of values in real space according to a sequence in real time. Once a level of understanding of the component and/or subcomponents has been reached, that is, a set of assertions have been

formulated which consist of values in the program's name space, set in a particular sequence, the analyst confirms the validity of those assertions with run-time information.

An analyst's facility for deriving assertions regarding the software component in relation to its computational history depends upon the individuality of the analyst, the feasibility of the application problem, the reliability of the computing environment, and the internal complexity of the software implementation. The debugging methodology is seen to work best with well-constructed software, given an experienced analyst, a feasible application problem in an accepted computing environment.

2.4.3 Hypotheses

In the scientific community, a hypothesis is an assumed proposition used as a premise in proving something else. It is an explanation for the occurrence of some phenomena and is made in order to test its logical or empirical consequences. In the debugging methodology, a hypothesis is formulated to explain the discrepancy between expected program behavior, as stipulated by assertions about program state values, and actual program behavior, as evidenced by run-time information. The hypothesis should be unambiguous, logical, testable and indicate predictable and verifiable results. It may also suggest possible changes to specified program components or work-around procedures that result in avoidance of the error. The role of a hypothesis in debugging is to propose a tentative explanation of the cause of an error. If the hypothesis can be validated, it becomes either the basis for the resolution of the error or, at least, a partial explanation of the error which directs further investigation. The error explained by the hypothesis may not be the error described in the SDR. In that case, the debugging process must be continued until a hypothesis can be formulated and tested which can explain the original error. If the hypothesis can not be validated, then the debugging process has to be continued until a hypothesis can be formulated and validated.

2.5 MANAGEMENT DATA

This study assumes that a body of information exists on the cost, schedule, and performance factors related to all the activities of the software development process. This information is generated and maintained as a result of normal management tasks of planning, monitoring and controlling a software development effort. While this information is not used directly by the debugging analyst, it does impact the debugging process. Debugging activities are performed within cost, schedule, and performance constraints just as are all other software development activities. The management decision on the length of time and/or resources used to isolate the cause of an error is dependent upon these factors as well as the criticality of the error and the availability of work-around procedures. The management decision on how to resolve an error that has been isolated is also dependent on these factors. These management issues are discussed in Section 3.3.5, Resolution/Termination.

3. THE DEBUGGING METHODOLOGY

This section describes a methodology to be followed for debugging software errors in integration-level testing/debugging. It provides a generic description of the activities required of a debugging analyst for isolating and resolving an observed software error. The debugging methodology presented is based upon a structured and disciplined approach to software development. Two necessary components of this approach which must exist for the debugging methodology to be applicable are:

- Existence of the following information components:
  - Application system descriptions.
  - Abstract machine descriptions.
  - Run-time information reflecting actual program performance.
- A hierarchically designed software system in which single functionality has been allocated to program components that, in addition, interact with each other in a well-defined manner, and the operation of one component does not depend upon the internal structure of another component for its correct performance.

Section 3.1 presents an overview of integration-level testing in order to provide the context in which debugging, as indicated by the methodology, takes place. Section 3.2 presents an overview of the flow of information within the debugging process. It is presented to clarify the information relationships. The debugging methodology is presented in Section 3.3 by way of a process model. This model defines a sequence of activities to be performed during each instance of debugging.

3.1 OVERVIEW OF INTEGRATION-LEVEL TESTING

The integration testing described in this report is part of computer programming test and evaluation (CPT&E). (Figure 2-2 depicts the Full-Scale Development Phase relationships with integration testing/debugging.) CPT&E tests are tests conducted prior to and in parallel with preliminary or formal qualification tests. CPT&E tests are oriented to support the design and development process and the formal acceptance testing--preliminary qualification tests (PQTs) and formal qualification tests (FQTs). As used in the context of this report, CPT&E testing includes, first, module-level testing and second, integration-level testing. Whereas module-level testing refers to the testing applied to a program component on a stand-alone basis, integration-level testing is testing applied to multiple program components. Integration-level testing may refer to combining sub-modules of a computer program component (CPC), or CPCs with CPCs, depending upon the definition and use of these terms by the development contractor. Further, integration-level testing, as referred to in this report, implies that the program components (i.e., modules and/or CPCs) are combined and tested on an incremental basis. Incremental integration testing focuses on:

- A sequential integration of functionally-related program components.
- Using outputs of one program component as inputs to the next.
- Verifying that the combined program components operate as designed and according to performance requirements.
- A planned and documented testing procedure which defines the relationship between inputs and required results.

The organized management of testing and debugging requires that a document such as a software discrepancy report be used to aid in controlling detection and resolution of the error. This documentation of software errors is essential when the testing responsibility is separate from the debugging responsibility. The software discrepancy report links the description of the observed error with an identification of the test configuration which together define the test environment which demonstrated the problem. (The software discrepancy report is described in Section 2.3.1.) The test configuration includes all that information which identifies the test cases and abstract machine, (i.e., hardware/software components) used during the test run. Status of the test configuration is necessary for problem duplication and resolution. (Section 3.4.3 of Volume III of this study discusses techniques for reconstructing the test configuration.)

Testing methodology is described in more detail in Section 3.1.7 of Volume III of this study. In addition, test plans and procedures are described in Sections 2.1.1 and 2.1.3 of this volume.

3.2 OVERVIEW OF THE INFORMATION FLOW

The methodology for software debugging is primarily based on the identification and use of the various structures of information needed to debug a software error. While the debugging process model attempts to offer a plausible view of the types of human thought activities needed in isolating and resolving a software error, it presents a rather primitive view of these activities. The information components upon which the analyst bases the problem solving activities are not primitive in nature, however, and are identified in relationship to the human thought activities. These information components, while generic in nature, must exist in one form or another for each particular system undergoing integration testing/debugging. Figure 3-1, Debugging Process/Information Components presents a detailed view of the flow of information (data) within the model. Figure 3-2 presents a simplified view of only the information components, and the manner in which they depend upon each other for their existence.

Existing before each instance of debugging are management information, the descriptions of the application software, abstract machines, and run-time information, specifically the software discrepancy report. These are all non-derived components of the debugging methodology, i.e., they are established during the software system life cycle before the need for debugging arises. During each instance of debugging, other information components are either derived by processes of the model which involve human understanding, or by use of software/hardware tools which collect and display information.

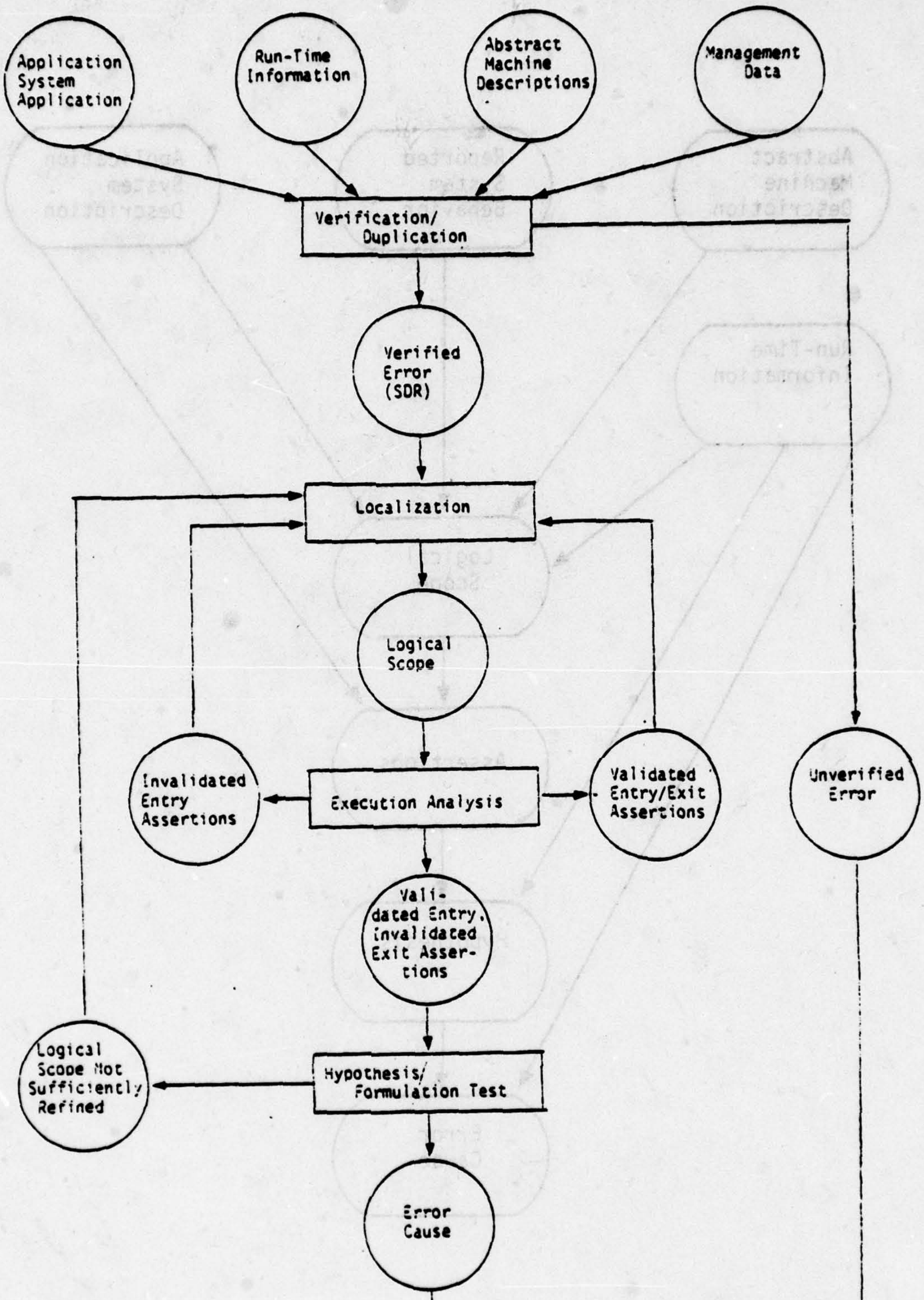


Figure 3-1. Data Flow in Debugging Process Model

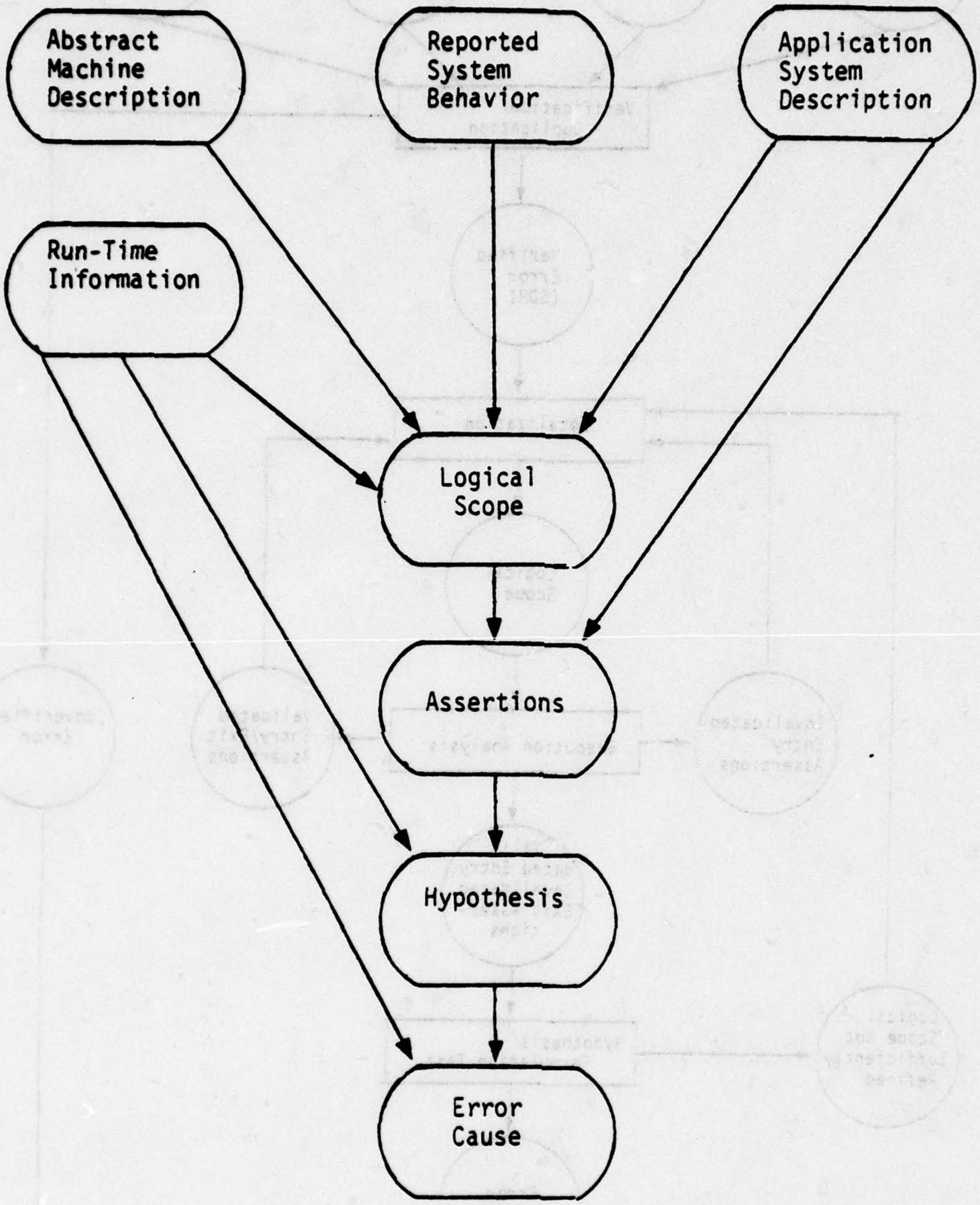


Figure 3-2. The Direct Dependent Relationship of Information Components of the Debugging Methodology

These derived information structures include run-time information, logical scope, assertions, hypothesis, and error cause. These information structures are briefly discussed below.

Probably the single most important body of information in the model is the run-time information. Although the debugging process is occasionally brought to successful termination by means of intellectual comprehension alone, it is more common for the analyst to require a body of data representing program execution at specific points in the computational history against which to test his intellectual comprehension. Run-time information comprises an information body which fulfills this purpose. As noted in Section 2, it consists of two components: error symptoms and program state information. It represents the accumulated body of information about the manner in which the target system reacts to its environment in terms of its external input and output interfaces. It is often necessary, while debugging a problem, to elicit system responses to a wider range of environmental parameters than were present when the problem was originally described. Run-time information, then, provides insight into the internal state of a program in the presence of a given external environment.

Run-time information is given in terms of values of program variables, historical sequences of program operations, and detailed information about the state of the abstract machine (i.e., operating system tables, storage maps, etc.) The two components of run-time information (i.e., error symptoms, program state information) are of different natures, and require different tools and techniques for their collection. Each is needed to support the creation of other bodies of information derived during the debugging process. The importance of run-time information is demonstrated by the fact that all other derived information components of the model depend upon it, as shown in Figure 3-2. (A useful adjunct to the debugging methodology would be an analysis of error symptoms and error causes in relationship to the concepts developed in this study. For example, more guidance could be given as to which information component should be examined at what time in relation to a set of recognizable error symptoms. This type of guidance might narrow the investigation for the bug, but it requires error data collection associated with hierarchical software development to be effective.)

Another significant information component used in the debugging model is the logical scope. Stated formally, the logical scope is the current node (i.e., a named functional component), in the hierarchical tree of abstract machines which implement a system, that is being investigated to determine its correctness according to specifications. Stated less formally, the logical scope represents that area of the software system which is currently being debugged in the process of back-tracking the computational history of the software to determine the anomalous condition. It may be defined by a quick flash of insight or a detailed analysis of all available information structures. In either case, the end result of an iteration of logical scope definitions is the isolation, or location, of the software error. Definition of logical scope depends upon the analyst's correct understanding of application system, its external interfaces with the abstract machine, the reported software performance and the resulting error. It also depends upon run-time information to provide insight into the manner of how a software design and source code implementation is flawed so as to produce results different than those required.

Two important concepts are associated with establishing the software boundaries represented by the logical scope. The first concept is that of definition of logical scope. This implies that the analyst is looking for the program component assigned to implement a specific functional area, as specified by a design, which demonstrated performance different from specified. Once the analyst has associated the earliest occurring software error with a functional component, the analyst refines the logical scope. This second concept, refinement of logical scope, implies that the analyst must examine the source code of the highest level program component and/or its interfaces with the abstract machine, or any program component in the hierarchy which assists it in performance of the functional capability until he has located the offending or missing construct(s).

The third derived information component is the set of assertions associated with a logical scope. On a global scale, these assertions represent the assumptions that lie behind the implementation of a design for a functional capability. On a more detailed level they are the values of a program's name space in a given point in time in response to a given set of input conditions. There are three types of assertions: (1) Entry assertions are those values of global data and input parameters which represent the assumptions a component makes about its environment when it is invoked. (2) Exit assertions are those values of data set by the module in the implementation of a given function at its termination point. (3) An intermediate assertion is an entry or exit assertion about the behavior of a component's sub-components.

The next information component, the hypothesis, is based on the assertions. The analyst formulates assertions in reference to a specific logical scope and tests their validity by comparison with run-time information. When the comparison of asserted values with actual values fails, the analyst has invalidated assertions for program values, provided by assertions, and actual program values, as indicated by run-time information. The hypothesis can attribute the discrepancy to any one or more of the existing information components: the application system description, the abstract machine description, the reported system performance, or the analyst's assertions. The creation of the hypothesis is an entirely intellectual process but, as modeled in this study, the hypothesis is based upon a discrepancy in the comparison of values asserted to hold true with information reflecting actual program performance.

The final derived information component is the error cause itself. This is implied by a hypothesis which has proven to be correct in light of the tests made on it by the analyst. It is the valid hypothesis, then which points to the resolution of the observed error. The manner in which the debugging methodology produces and uses the information components of the development process to isolate and resolve a software error is presented in the following section.

3.3 THE DEBUGGING PROCESS MODEL

This section presents the activities a debugging analyst must perform in isolating and resolving a software error. Many of these activities are intellectual activities which are highly individualistic in nature. The purpose of the debugging process model is to identify and suggest a sequence of debugging activities which may better organize the usual ad hoc approach taken in integration-level software debugging. Many debugging activities in the process model are actually not distinct steps since they are intellectual activities and may be performed almost intuitively and/or instantaneously. Further, the process model does not mean to suggest that the debugging methodology which it presents will necessarily solve all debugging problems. Lastly, the debugging process should be considered iterative in nature, some processes requiring computer usage and, perhaps, days in time, other process iterations occurring in seconds, or milliseconds.

The debugging methodology is presented by way of a directed graph depicted in Figure 3-3. A directed graph is a set of nodes (represented as named "places") and a set of edges connecting those nodes (represented as arrows connecting nodes). The nodes of the directed graph are the activities required in debugging; the edges represent the possible alternatives that exist in selecting debugging activities. The debugging model describes the function of each activity, its particular use of the information requirements, its relationship to other activities, and a generic list of tools which can be used to augment or produce information components. (These tools are described in more detail in Appendix A.) Also included in the description of debugging activities are the assumptions made for each activity and the probable sources for errors or problems.

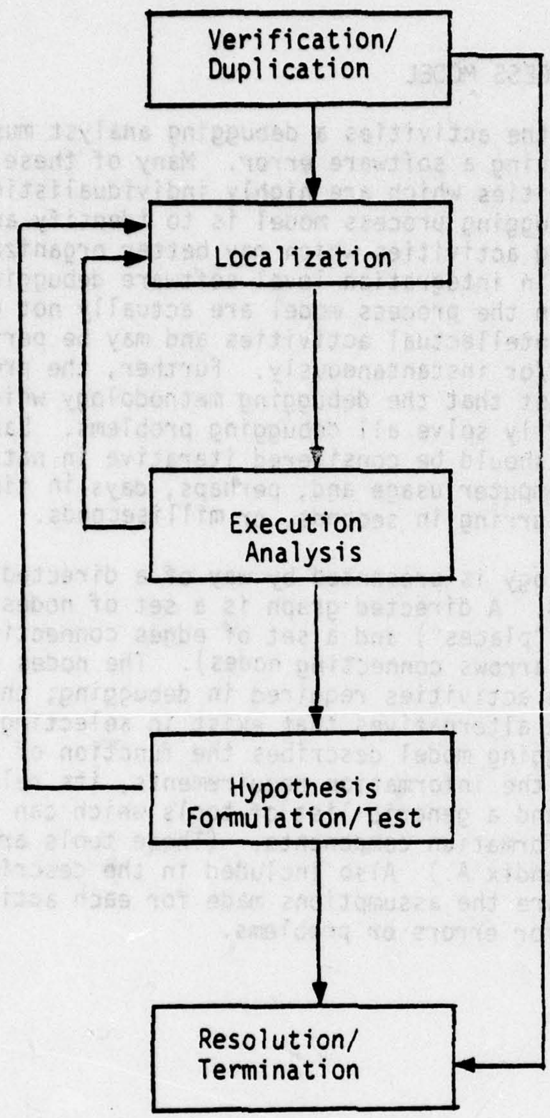


Figure 3-3. The Debugging Process Model

3.3.1 Verification/Duplication

Verification/duplication is the first process in the debugging methodology. It is initiated on receipt of a software discrepancy report (SDR), identifying a program behavior that differs from what was expected. During the verification/duplication process, the analyst attempts to establish that there is, indeed, an error and that the anomalous behavior is attributable to the software rather than to a human/machine error such as an incorrect hardware/software configuration, an invalid expectation of program behavior, an erroneous interpretation of required inputs or outputs, etc. It may not be possible, at this point in the debugging process, to establish conclusively that all problems are caused by software failure. Other factors, such as machine failures, are isolated in the same manner as software errors.

To verify that there is a software error, the analyst needs valid information about the hardware/software configuration, test inputs, program behavior, and program state information at the initiation of the software's operation and at the time the problem occurred. If all the required information is not supplied with the SDR, or if the analyst has reason to suspect the SDR's accuracy, he obtains the necessary information by contacting the person who reported the problem and by duplicating the test run in which the problem occurred. In duplicating the problem, the analyst must be careful to duplicate the exact conditions (to the extent possible) which caused the anomalous behavior, without introducing debugging tools which were not present in the original run since introducing them may alter the program's behavior. If, however, after duplicating the problem, the analyst finds he has inadequate information, he may then rerun the test and include the necessary tools.

The analyst may or may not be successful in duplicating the problem. Failure to duplicate the problem may be due to insufficient or incorrect information about the hardware/software configuration, test inputs, or program state, or to an intermittent hardware/software failure. If duplication of the anomalous behavior is not possible within resource constraints, a management decision is required to resolve the problem. (See Section 3.3.1.6.)

After obtaining the necessary information, the analyst reviews it to gain an understanding of the actual program behavior evoked by the test inputs and the initial program state. He then reviews the system design and the requirements to gain an understanding of the required program behavior for the given inputs and initial program state. Even with program behaviors that are obviously in error (as, for example, an infinite loop), it is wise for the analyst to determine the required program behavior as an aid to debugging the anomalous condition.

Depending upon the nature of the cause of the error and the error symptoms it generates, the analyst may verify, after comparing the actual and required program behavior, that a discrepancy actually exists. Its isolation and resolution may require an in-depth analysis or it may be an obvious logical flaw which is readily isolated and easily repaired. This methodology is not needed nor intended for the analysis and isolation of trivial software errors. It assumes that the majority of these types of errors have been removed during module-level testing. If error symptoms are recognizable and easily mapped to a faulty program component, many of the steps in the following processes can be omitted. Therefore, after verifying that a discrepancy

exists, the analyst proceeds to the next process in the debugging methodology, localization (See Section 3.3.2). Otherwise, he reports his findings and terminates the debugging process.

The steps in the verification/duplication process are depicted in Figure 3-4. The process is described in greater detail in the subsections that follow.

3.3.1.1 Process Steps

The steps in the verification/duplication process are:

- Duplication. This step is optional and can be omitted if sufficient valid information is supplied with the SDR. If additional information must be collected, the analyst does the following:

    - Reconstruct the operational status of the abstract machine by duplicating both the hardware/software configuration and the operational status of the external interfaces at the time the error was observed. The more complex the abstract machine's external interfaces, the more difficult it is to reconstruct the exact configuration.

    - Reconstruct the operational status of the application software by duplicating the exact software test configuration which exhibited the anomalous behavior.

    - Reconstruct the test environment by duplicating the test input state and initial conditions which evoked the reported error.

  If the analyst is unable to duplicate the problem within resource constraints, the problem is resolved by a management decision.

- Understanding. The analyst acquires an understanding of the actual program behavior by examining the information supplied by the SDR or collected as a result of duplication. The analyst acquires an understanding of the required program behavior for the given inputs and initial program state by examining the requirements and/or design specification. The analyst acquires an understanding of the abstract machine (i.e., hardware/software configuration) used by examining the supporting documentation. The level of understanding gained should be commensurate with the level needed to verify that a software error exists.

- Symptom Accumulation. If all of the information required by the debugging analyst to verify the existence of a software error is not supplied by the SDR, he may have to accumulate additional problem symptoms and run-time information. The analyst gathers error symptoms by examining the program's run-time external performance or behavior. He gathers program state information at the initialization of the run and after the occurrence of the error by the use of debugging tools that do not perturb the program's behavior.

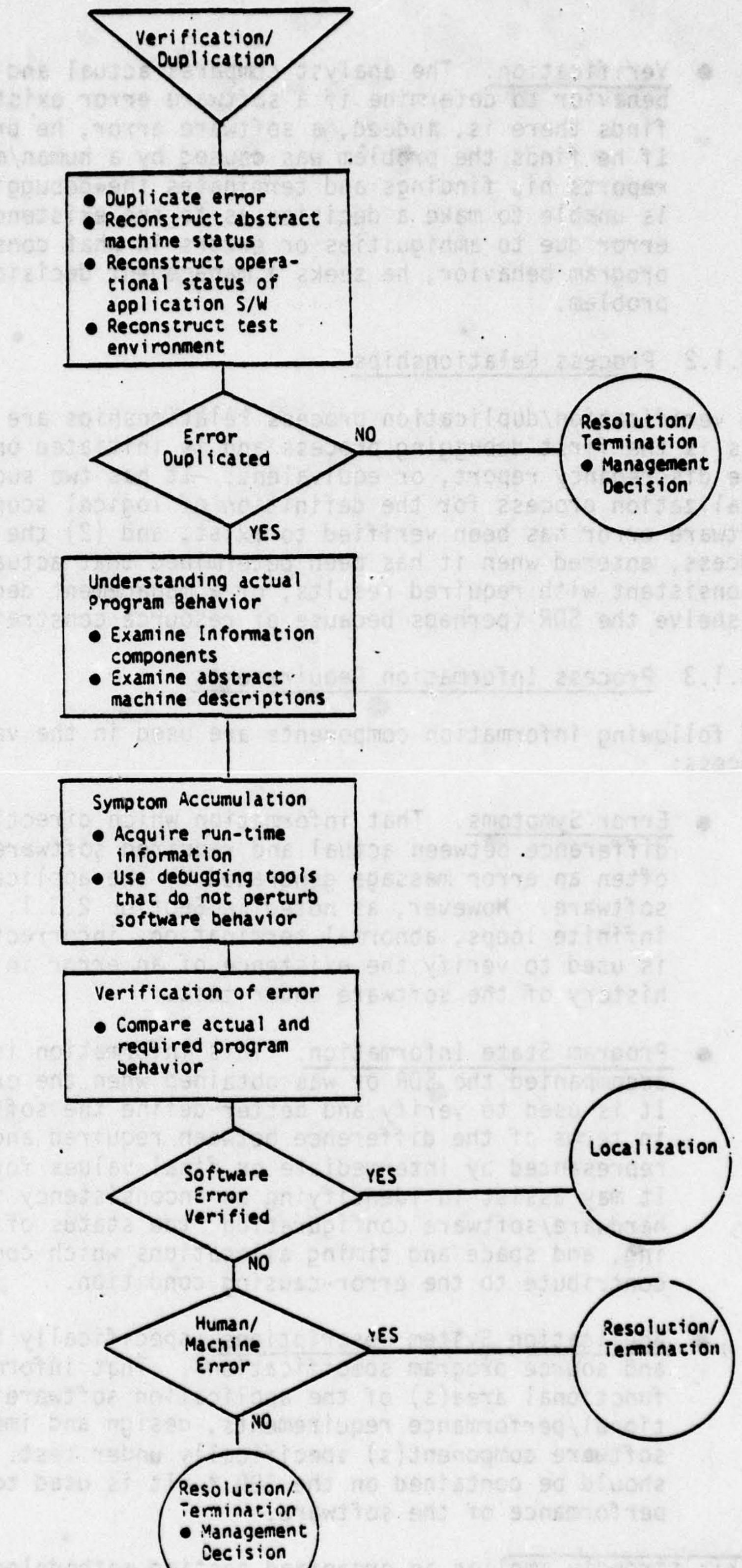


Figure 3-4. Verification/Duplication Process

- Verification. The analyst compares actual and required program behavior to determine if a software error exists. If the analyst finds there is, indeed, a software error, he proceeds to debug it. If he finds the problem was caused by a human/machine error, he reports his findings and terminates the debugging process. If he is unable to make a decision as to the existence or cause of an error due to ambiguities or errors in what constitutes required program behavior, he seeks a management decision to resolve the problem.

3.3.1.2 Process Relationships

The verification/duplication process relationships are shown in Figure 3-4. This is the first debugging process and is initiated on receipt of a software discrepancy report, or equivalent. It has two successors: (1) the localization process for the definition of logical scope, entered when a software error has been verified to exist; and (2) the resolution/termination process, entered when it has been determined that actual results are not inconsistent with required results, or a management decision has been made to shelve the SDR (perhaps because of resource constraints).

3.3.1.3 Process Information Requirements

The following information components are used in the verification/duplication process:

- Error Symptoms. That information which directly relates the observed difference between actual and required software performance. It is often an error message generated by the application/operating system software. However, as noted in Section 2.3.1, it can also be infinite loops, abnormal termination, incorrect outputs, etc. It is used to verify the existence of an error in the computational history of the software under test.

- Program State Information. This information is available only if it accompanied the SDR or was obtained when the error was duplicated. It is used to verify and better define the software error, especially in terms of the difference between required and actual behavior, as represented by intermediate or final values for program variables. It may assist in identifying an inconsistency in the status of the hardware/software configuration, the status of simultaneous processing, and space and timing allocations which could account for or contribute to the error-causing condition.

- Application System Descriptions, specifically the requirements, design and source program specifications. That information relating to the functional area(s) of the application software which specify functional/performance requirements, design and implementation for the software component(s) specifically under test. This information should be contained on the SDR.* It is used to determine the required performance of the software.

* This strongly implies an organized testing methodology which is based on incremental integration testing. In addition, test cases used are clearly identified and based on testing component functionality and interfaces, if not also on internal control structures.

- Abstract Machine Descriptions. That information which describes the external interfaces of the application software with the software/hardware configuration components. It is used to understand the functional components of the abstract machine.

- Management Data. That information which is assumed to exist in all software development projects which have cost, performance and schedule contractual constraints. It is used in deciding on the action to take when this process has been unsuccessful in verifying and/or duplicating the software error.

3.3.1.4 Tool Usage

Verification of the software error may require duplicating the conditions which caused its occurrence. This sometimes can be difficult, if not nearly impossible, task. Generally, if the original testing procedures are methodically controlled, the debugging analyst will have a high degree of confidence in the original SDR and duplicating the software error is not necessary at this point. However, if it is necessary to duplicate the error, there are available, in a well-structured development environment, a number of tools for aiding the reconstruction of the exact test case, and hardware/software configuration including the data base and/or initial conditions. They include: *

- Program Support Library
- Test Input Listing Tool (See Appendix A, Section A.24)
- Script Tape (See Appendix A, Section A.24)
- System Status Summary Tool (See Appendix A, Sections A.26, A.27)
- Checkpoint/Restart (See Appendix A, Section A.28)

Verification of the software error may also require the gathering and analysis of run-time information by special debugging tools. These tools are used to obtain initial program state information and that information which is available after the occurrence of the error. Those tools are used in other processes of the debugging process model, but in the duplication/verification process, care should be taken to use tools which will not perturb the execution of the configuration of the system in response to a specific test case in such a way that the error will not be repeatable. Tools that operate during the program's operation, such as snapshot dumps, breakpoint/trap dumps, or trace program, should not be used. Tools that can be used include:**

- Post-mortem dumps (See Appendix A, Section A.1)
- Auxiliary dump (See Appendix A, Section A.6)

* Further discussion of these tools can be found in Section 5 and Appendix A, and Section 3.4.3 of Volume III.

** Further discussion of these tools can be found in Section and Appendix I, and Section 3.3.2 of Volume III.

3.3.1.5 Assumptions Made

This process, as all other processes in the model, assumes that the information requirements which are described for each process exist and are understandable to the debugging analyst. Obviously, if all of the information is not available, the analyst must verbally communicate with other development project personnel. Even when the project environment supports the development process in a well-structured manner so that documentation is available and accessible, complex debugging problems sometimes require interaction with hardware engineers and operating and support software system personnel.

Another assumption made by this and other tool using processes is that the analyst be able to effectively use the debugging support software to collect run-time information (i.e., error symptoms and program state information) and interpret it for debugging analyses.

3.3.1.6 Error Sources/Problems

A primary problem which can be encountered in this process is the failure to verify/duplicate the reported problem. The software discrepancy report may not accurately describe the software problem; the requirements specification may be amgibuous; the error may be manifesting itself intermittently; etc. When a software error is known to exist intermittently, a management decision regarding error resolution must be made based on cost, schedule, and performance contraints (see Section 2.5). This is necessary because duplication of the program state may require expenditure of a large amount of resources since it is sometimes complicated by intermittent hardware errors, CPU and I/O device timing variations, inability to replicate the exact software processing of the abstract machine, etc.

A problem may be difficult to reproduce if the SDR does not include sufficient information about the test, environment, and application system at the time of the error, or if the SDR is in error (i.e., factors which the test team believed to be involved may have been different than those actually involved). In theory, if a complete specification of the input space vector* is provided, sequential (as opposed to concurrent) programs exhibit perfectly reproducible behavior. In the real world, such complete specification of the environment is rare and much effort may be expended to reconstruct the real environment.

Systems which include concurrent processes may exhibit behavior which is not comprehensible in terms of the source specification of the functional processes. This can be due to several reasons. For example, if the operations of the abstract machine are not divisible when executed by the underlying machine, time dependent behavior can occur. Problems may arise due to small differences in the relative speed of concurrent processes which can alter program timing

* An input space vector is the totality of data values required to initiate the operation of the software in question. It includes all the data values set by external or interfacing hardware/software components.

to a sufficient degree to change program behavior. The basic problem involved is that the behavior of concurrent processes which share data generally depends on the order in which the process operations are performed. Unlike purely sequential programs, in which the order of execution of process operations is uniquely determined by input data, a concurrent program may exhibit non-reproducible behavior. It is difficult to duplicate a problem when such an error occurs, and it is necessary to recreate the sequence of events involved.

3.3.2 Localization

During the localization process, the analyst defines the logical scope, i.e., the boundaries within the software, in which he suspects the error occurred and on which he will concentrate his efforts. To define the logical scope, he first examines the error symptoms, as reported in the SDR or gathered during problem duplication (see Section 3.3.1), and any additional run-time information he may have generated, looking for an insight as to what caused the error. Then the analyst examines the design specification supported, perhaps, by cross reference information and requirements traceability matrices, to determine candidate logical scopes for where the error could have been caused. Depending on the type of error symptoms that exist, the analyst may find it necessary to run additional tests to gain further insight into the conditions under which the error manifests itself or to otherwise eliminate logical scopes until he has selected a single one. The selection of a logical scope may, at times, be a spontaneous act, driven by a quick insight. But, at other times, it will be a laborious task requiring a thorough investigation by the analyst.*

The localization process is iterative in nature as it is the mechanism used to isolate the boundaries of the specification which contains the error. The analyst first broadly defines the functional area in the software which is implicated by the difference between required and actual results as indicated by the test case, test procedure, and related design specification. Depending upon the size and complexity of the software under test, the analyst's familiarity with it and the associated error symptoms, the analyst may, at times, be able to easily define a logical scope within the source program representation which appears to be malfunctioning. At other times, the analyst must refer to the design specification related to the functional capability under test in order to formulate and test global assertions for a high-level logical scope which will then be iteratively refined as the scope of inquiry narrows. The intent for defining or refining the logical scope is to bound the focus of investigation during debugging according to a logical scheme which is based on examination of the computational history of the

* It is seen that a correlation of error causes with error symptoms would be a valuable mechanism by which to aid in formulating the logical scope. In addition, such a correlation might indicate that there is specific need to examine a particular system description representation instead of each of them. As noted earlier, the correlation of error symptoms/error causes, especially in relation to a hierarchically developed software system, has not been performed as a part of this study and such guidance is necessarily omitted.

software under test. Once a logical scope has been defined/refined, the analyst formulates and tests assertions regarding the logical scope in the execution analysis process (see Section 3.3.3). The purpose for the iterative definition/redefinition of logical scope is to isolate logical scope containing the error, (i.e., one in which no error existed on entry but did exist on exit). Eventually, the iterations result in identification of the program component, or construct within, which causes that component, or one in its hierarchy, to function in such a manner as to cause its ouput to be incorrect when processed as input by an interfacing component.

The localization process is similar to the process of hierarchical development. The analyst first looks at the specifications of the functional capability at a global level, then refines that scope until the implementation is found to be either correct or incorrect. If the implementation is found to be correct, the analyst must "debug" the design or requirements, a task which involves verification activities of another nature. In this process, as in all others, the analyst first assumes that the anomalous condition is an error in implementation, i.e., in the translation of the design into the source program representation. Only when the implementation is not found to be in error, does the analyst consider errors in the design or in the abstract machine.

There may be times when the analyst will not select a logical scope during localization. Numberous reasons exist for this decision. For example, it may be that the analyst realizes he made an error during the execution analysis process and decides to reenter it. Or he may be convinced, either on examination of additional information or intuitively, that the error resides in the selected logical scope and decides to reattempt the execution analysis process. There are also times when the analyst may select a previously examined logical scope, as when subsequent attempts seem to lead him around in a circle or when he has lost the trail of the error. Regardless, a methodical examination of the components in an hierarchy of programs and sub-programs based on the validity of entry and exit assertions should lead the analyst to a logical scope in which no error existed on entry but did exist upon exit, either by an error in implementation of the design or a deficiency of the design to meet the requirements.

The steps in the localization process are depicted in Figure 3-5. This process is described in greater detail in the subsections that follow.

3.3.2.1 Process Steps

There are two steps within localization process, both of which may need to be supported by an additional data collection step. These steps are described below.

- Define Logical Scope. This step is performed on initial entry to localization. It is also performed each time the execution analysis process (see Section 3.3.3) fails to confirm that the error resides in the previously selected logical scope. The analyst uses, as input to the definition task, the original error symptoms and all supporting run-time information. If the analyst has located a logical scope in which the error existed on entry, he uses the invalidated entry assertions, the error symptoms, and the computational history as additional inputs. Then he examines the design specification to

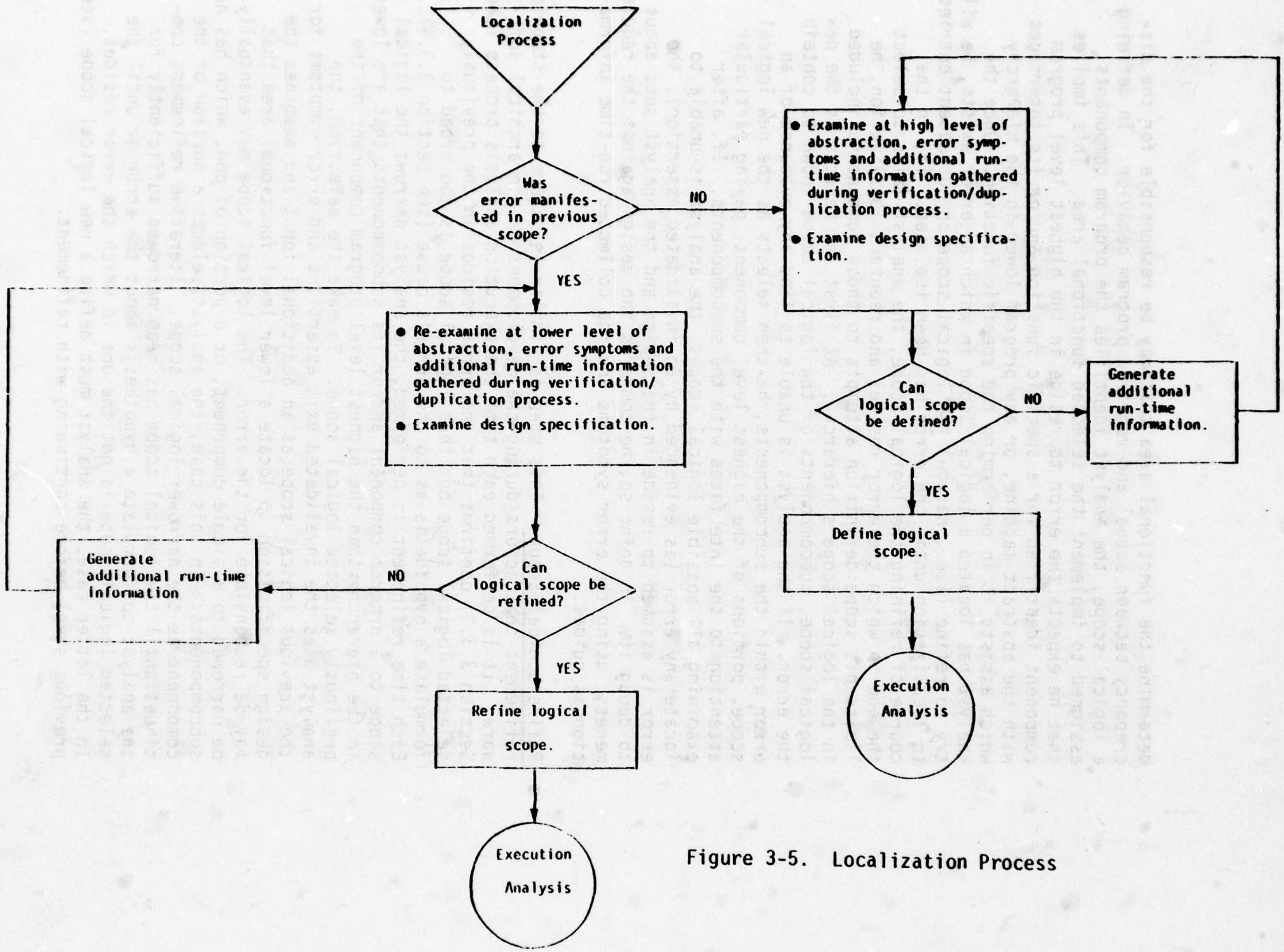


Figure 3-5. Localization Process

determine the functional area that may be responsible for the discrepancy between actual and required program behavior. In defining a logical scope, the analyst identifies the program components assigned to implement the selected functional area. This implies that he expects the error to reside in the highest level program component identified for a specific function and/or its interfaces with the abstract machine, or any program lower in the hierarchy which assists it in performing the specific function. Once the analyst has located a logical scope in which an error exists, he will try to refine (i.e., narrow) the logical scope on subsequent entries to localization until the error has been isolated. If, in the course of refining the logical scope, the analyst fails to select the one in which the error resides and reenters localization, he limits his scope definition attempts to those components included in the logical scope's hierarchy. At first, he selects as the new logical scope, subcomponents of the logical scope known to contain the error. If the analyst is unable to find any evidence of an error within the subcomponents, he then selects as the new logical scope, portions of the highest level component, paying particular attention to the interfaces with the subcomponents. If, after examining all possible logical scopes, the analyst is unable to locate any error (as evidenced by an invalidated assertion), the error is assumed to reside in the design and the analyst sets about to debug it. In doing so, he compares the design against the requiremenets, using the error symptoms and the collected run-time information as guides.

- Refine Logical Scope. This step is similar in nature to the stepwise refinement that occurs during the development of hierarchical software. It is performed each time the execution analysis process (see Section 3.3.3) confirms that the error resides in the previously selected logical scope but that logical scope is too broad to formulate a hypothesis as to the error's cause (see Section 3.3.4). Each time refinement is performed, the analyst narrows the logical scope to a program component and/or its subcomponents that are lower in the hierarchy than the highest level program component of the previously selected logical scope. To make the selection, the analyst uses the invalidated exit assertions and error symptoms for the previous logical scope as an additional input and examines the design specification to locate a lower level functional area that may be responsible for the error. The logical scope may eventually be narrowed to a single component, or a portion of one, which has no subcomponents. In this case, the analyst selects a portion of the component as the narrower logical scope. Iterative refinement continues until the logical scope has been narrowed sufficiently for the analyst to formulate a hypothesis about the error or until the selected logical scope is not the one in which the error resides. In the latter case, the analyst must define a new logical scope (see previous step) before continuing with refinement.

- Determine Computational History. The definition/redefinition of a logical scope must be based on the computational history, i.e., the actual operational sequence among the program components in response to the test inputs and initial conditions which caused the error to be manifested. Depending on how knowledgeable the analyst is with regards to the software under test and how successful he has been in defining the logical scope for investigation, he may not be able to determine the operational sequence by examining static hierarchy of program components described in the design and source language specifications. If this is the case, it will be necessary for him to collect dynamic flow information by using tools that selectively trace the program's execution.

3.2.2.2 Process Relationships

The localization process relationships are shown in Figure 3-5. This process is the second debugging process and it starts the iterative activities needed to isolate this software error. It is entered initially after verification/duplication to establish the initial boundaries on which to focus the analysis of the next processes. Subsequent entries from execution analysis and hypothesis formulation/test are made to refine the previous logical scope or define a different one. The localization process exits to execution analysis process each time a logical scope has been defined, or refined.

3.2.2.3 Information Requirements

The following information components are used in the localization process for defining the logical scope:

- Run-time Information, including:

  - Error Symptoms. That information which indicates an erroneous computational history in response to test inputs and initial conditions.

- Application System Descriptions, specifically the requirements, design and source program specifications. That information relating to the functional components of the software under test which manifested program performance which was different from that required.

3.2.2.4 Tool Usage

This process uses tools to help define or refine the logical scope. Definition of the initial logical scope can be aided by tools that relate error symptoms to functional program segments. For example, the Set-use Matrix/Cross Reference analysis tool (see Appendix A, Section A.10) can be used to identify all areas of the program where an error message is referenced or where anomolous data is set. This type of tool provides a static representation of the setting and using of all name-space values, which gives the analyst the first approximation of where erroneous processing can occur.

The analyst will also use tools which capture and display information relating to the operational sequence of program segments. While the entire computational history of the software can be recorded by traces, the analyst must be selective about such requests. The objective for the use of tools in this process is to support back-tracking analysis by displaying a selective area in the software during execution in order to determine how a given computation was influenced by previous computations. Tools which provide this information include:

- Snapshot Dumps (See Appendix A, Section A.2)
- Programmed-In Dumps (See Appendix A, Section A.4)
- Trace Routines (See Appendix A, Sections A.7, A.8, A.9, A.15)
- Program Flow Analyses (See Section 3.1.8.1, Volume III)

3.3.2.5 Assumptions Made

The definition and refinement of the logical scope as presented in this process assume a hierarchical software development which is well documented in the application system descriptions. A second assumption made is that requirements are directly traceable to both the design and the implementation of functional components.

3.3.2.6 Error Sources/Problems

Given that the assumptions stated in Section 3.3.2.5 hold true, the principle source of problem in this process is determining the computational history of the software. The use of tools which provide dynamic control flow and dependency will alleviate some of this problem if they are available in the computing environment.

3.3.3 Execution Analysis

This process is entered from the localization process for the purpose of determining the correctness of the logical scope which localization produced. It is characterized as being a human thought process which is distinctly individualistic in nature. It is rudimentarily characterized in this methodology as the process of formulating and validating/invalidating assertions about the expected values of interfacing parameters and global data on entry to and exit from the selected logical scope. These assertions are made in light of the required performance in terms of a specific implementation. Depending upon where the analyst is in the back-tracking of the computational history and the nature of the error symptoms he is examining, the assertions being formulated may address the functionality of a high-level component or a particular control structure within a component.

Once a set of assertions has been formulated, the analyst compares actual program performance data with asserted values for program variables to ascertain if the program's operation is correct as he understands it. To do this, the analyst validates or invalidates the sets of assertions formulated. He does this by determining the actual values of the pertinent parameters and global data by examining existing run-time information or by generating any required additional run-time information. If the expected and actual values compare on entry to the logical scope (i.e., validated assertions) but fail to compare on exit (i.e., invalidated assertions), the analyst has located the logical scope in which the error resides and he proceeds to formulate a hypothesis that explains the cause of the error (see Section 3.3.4). If, however, the two sets of values do not compare on entry or compare on both entry and exit (i.e., the error occurred prior to entry or no error occurred), the analyst has failed to isolate the logical scope containing the error and reenters the localization process (see Section 3.3.2) to define a new one.

It should be recognized that the formulation of assertions is not an easy task. The resulting assertions may be too weak (i.e., too general) to detect a subtle error. They may be incomplete in that they do not address all the variables which affect or are affected by all the components in the logical scope. And they may be incorrect since assertion formulation tends to be an error prone activity.

The steps in the execution analysis process are depicted in Figure 3-6. This process is described in greater detail in the subsections that follow.

3.3.3.1 Process Steps

The steps in the execution analysis are described below. Those steps which use run-time information may involve an additional step, symptom accumulation, for data collection.

- Formulate Entry Assertions. The analyst formulates assertions about the expected values of input parameters and interfacing global data on entry to the logical scope. These assertions, known as entry assertions, are generally derived from the design and source language specification descriptions of the highest level component in the logical scope. These assertions need only address the expected

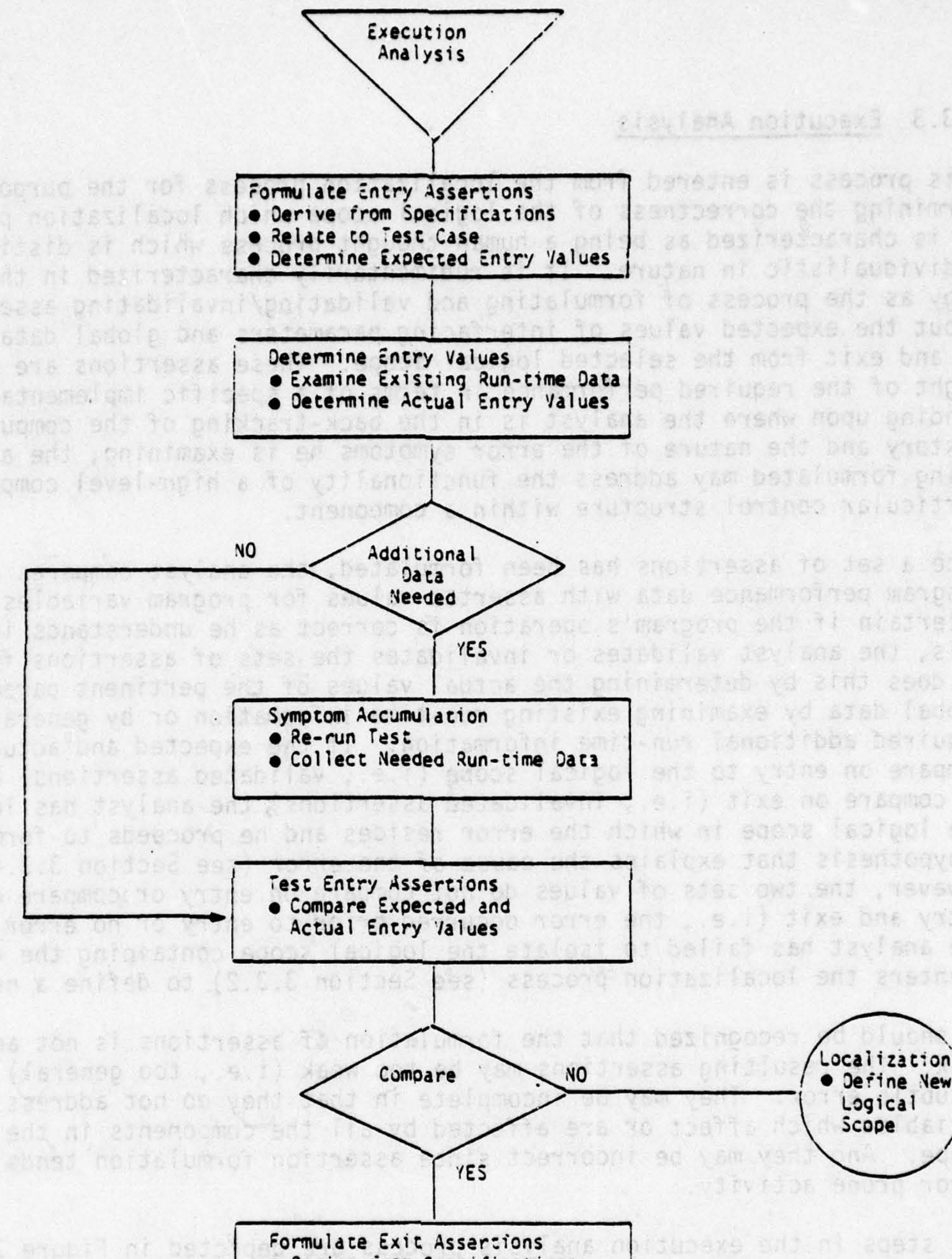


Figure 3-6. Execution Analysis Process

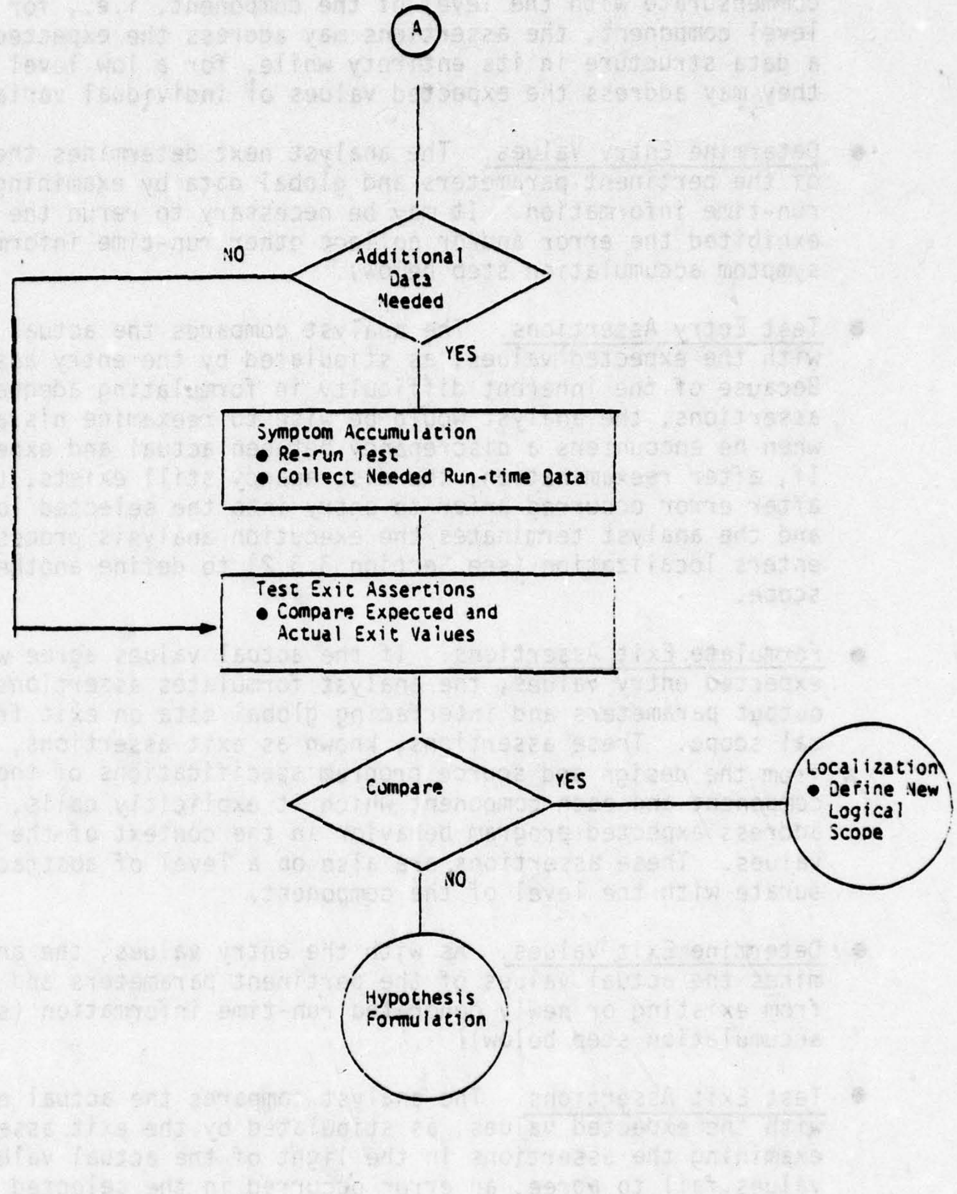


Figure 3-6. Execution Analysis Process (Cont'd)

program state in the presence of the input data defined by the current test case. The assertions are on a level of abstraction commensurate with the level of the component, i.e., for a high level component, the assertions may address the expected values for a data structure in its entirety while, for a low level component, they may address the expected values of individual variables.

- Determine Entry Values. The analyst next determines the actual values of the pertinent parameters and global data by examining existing run-time information. It may be necessary to rerun the test that exhibited the error and/or collect other run-time information (see symptom accumulation step below).

- Test Entry Assertions. The analyst compares the actual entry values with the expected values, as stipulated by the entry assertions. Because of the inherent difficulty in formulating adequate and valid assertions, the analyst would be wise to reexamine his assertions when he encounters a discrepancy between actual and expected values. If, after reexamination, the discrepancy still exists, the sought-after error occurred prior to entry into the selected logical scope and the analyst terminates the execution analysis process and re-enters localization (see Section 3.3.2) to define another logical scope.

- Formulate Exit Assertions. If the actual values agree with the expected entry values, the analyst formulates assertions about the output parameters and interfacing global data on exit from the logical scope. These assertions, known as exit assertions, are derived from the design and source program specifications of the highest level component and each component which it explicitly calls, and need only address expected program behavior in the context of the actual entry values. These assertions are also on a level of abstraction commensurate with the level of the component.

- Determine Exit Values. As with the entry values, the analyst determines the actual values of the pertinent parameters and global data from existing or newly generated run-time information (see symptom accumulation step below).

- Test Exit Assertions. The analyst compares the actual exit values with the expected values, as stipulated by the exit assertions, re-examining the assertions in the light of the actual values. If the values fail to agree, an error occurred in the selected logical scope and the analyst enters the hypothesis formulation process (see Section 3.3.4) because the anomalous condition has been isolated. If, however, the exit assertions are validated by the run-time information, the error occurred in a different logical scope and the analyst reenters localization to define it.

- Symptom Accumulation. The analyst may need to re-run the test that exhibited the error in order to generate actual entry or exit data that can be compared with the expected values. The actual data consists of parameters and global data and is obtained by special debugging tools which collect and analyze program state information.

3.3.3.2 Process Relationships

The execution analysis process relationships are shown in Figure 3-6 . This process is entered from localization during the iterations required to isolate the program component containing the error. The successor to this process is either: (1) the localization process for the redefinition of the logical scope; or (2) the hypothesis formulation/test process entered when the logical scope is found to contain the error and the analyst attempts to formulate a hypothesis as to its cause.

3.3.3.3 Information Requirements

The following information components are used in the execution analysis process:

- Logical Scope. Used as the definition of boundaries in the software which is being analyzed to determine its correctness.

- Application System Description, specifically the design and program source specifications. Used to support the formulation of entry and exit assertions associated with the logical scope.

- Run-Time Information.

  - Error Symptoms. Used to help determine the variables about which assertions are formulated.

  - Program State Information. Used as a source of actual entry/exit values which are compared with the asserted (or expected) values.

3.3.3.4 Tool Usage

Tools are used in both the formulation of entry/exit assertions and in the generation of run-time entry/exit values. There are no tools that can help the analyst to determine the expected values for the entry/exit variables; he must determine those by inspecting the specifications. However, a set-use matrix/cross reference analysis tool (see Appendix A, Section A.10) can help the analyst determine the variables about which assertions need to be made. The analyst first determines which software components are contained within the selected logical scope. He then uses the set-use matrix to determine the entry and exit variables of the software components.

There are a number of debugging tools that gather run-time information and help the analyst determine the actual values of entry/exit variables in response to a given test case. These tools collect and analyze program state information. They include the following:

- Post-Mortem Dumps (see Appendix A, Section A.1)
- Snapshot Dumps (see Appendix A, Section A.2)
- Breakpoint/Trap Dumps (see Appendix A, Section A.3)
- Monitor Dumps (see Appendix A, Section A.5)
- Dynamic Internal Trace (see Appendix A, Section A.7)
- Recorded Trace (see Appendix A, Section A.8)
- Monitor Trace (see Appendix A, Section A.9)
- Software Breakpoint/Trap (see Appendix A, Section A.16)
- Hardware Breakpoints (see Appendix A, Section A.17)
- Reversible Execution/ Backtracking (see Appendix A, Section A.18)
- Program Execution-Oriented Recording/ Reduction (see Appendix A, Section A.22)
- Input/Output-Oriented Recording/Reduction (see Appendix A, Section A.23)

Sometimes it is not practical to re-run an entire test case. The following tools can be used to input values for a given set of variables:

- Interactive Modification Tools (see Appendix A, Section A.19)
- Hardware Modification Tools (see Appendix A, Section A.21)

3.3.3.5 Assumptions Made

A primary assumption of this process is that a logical scope has been defined by the localization process. This implies that a software component (at either a high or lower level) has been selected as the one containing the error. A second assumption for this and all other processes is that the analyst has available and understands the application system descriptions needed in this process. High quality system descriptions are especially important if implementation type errors have been tentatively eliminated and design errors are under investigation. If an implementation error has been tentatively eliminated, the analyst will have to trace the allocation of functional requirements to the design specification. This allocation may not be clearly or consistently described in the design specification (and related documentation) or may have been misinterpreted by the designer. Even when project management ensures that documentation is available and accessible, the debugging analyst may have to interact with design personnel to clarify his understanding of their contents.

3.3.3.6 Error Sources/Problems

The formulation of assertions is one of the most difficult and error prone procedures in the debugging process. Deriving entry/exit assertions concerning the expected values of parameters and interfacing global data is a complicated mental process. The analyst analyzes the design and source language specifications and must determine expected program state values that would result from the input data defined by a specific test case in terms of time and space dimensions. The analyst, in effect, "simulate" the abstract machine to determine expected values resulting from program execution. Often he cannot simulate the performance of the abstract machine, and instead of determining exact values, he establishes an expected range of values. This range may be too wide or global, and may include values which are not correct. During the assertion testing processes, an incorrect actual value may be within the range asserted to hold true which results in the assertion being incorrectly validated. Conversely, an asserted range of values may be too narrow, and the assertion can be incorrectly invalidated. Obviously, a third problem can arise when the analyst cannot adequately establish entry/exit assertions.

In principle entry/exit assertions can be derived for a component without knowledge of its internal structure (e.g., source listing). Such assertions should be provided as part of a detailed design specification; however, this is not common in practice. Additionally, assertions provided as part of the design specification may be inadequate; i.e., they may be incorrect, overly weak, or not directly testable. As a result, a debugging analyst must often generate assertions from <u>both</u> his knowledge of the principles motivating the design and inspection of the source program specification for the component.

In the context of formal verification of program components, assertions are not dependent upon specific input data combinations; rather they are static statements of relationships between variables which hold for all inputs. As used in the debugging methodology, assertions may be input data dependent. While such assertions will be useful for determining exactly what went wrong for a specific test case, they generally will not provide insight into candidate solutions which prevent the introduction of new errors. The prevention of the introduction of new errors into a system while fixing old errors must be viewed as a currently unsolved problem which this study does not address.

3.3.4 Hypothesis* Formulation/Test

This process is entered after the analyst has confirmed that an error exists in the selected logical scope. In particular, the analyst has established, as a result of generating and testing assertions during the execution analysis process (see Section 3.3.3), that the entry assertions associated with the logical scope appear correct in light of the given test inputs but the exit assertions have been invalidated by run-time information. During the hypothesis formulation/test process, the analyst attempts to offer a plausible explanation for the invalidated assertions, i.e., the discrepancy between expected program

* According to Webster, a hypothesis is a tentative assumption made in order to draw out and test its logical or empirical consequences. As used here, it is a tentative explanation of the cause of an error.

values, as stipulated by assertions, and actual program values, as evidenced by run-time information. While it is hoped that this discrepancy is wholly responsible for the anomalous condition being debugged, it must be recognized that the condition may have been caused by other errors in addition to the located one, or that the located error is unrelated to the reported one.

The formulation of a hypothesis is a human thought process in which the analyst attempts to derive the specific cause of the error by examining previously accumulated, as well as specially-generated, run-time information and by examining the design and source language specification. The analyst looks for answers as to which operational conditions cause the error to manifest itself, where within the logical scope does the error occur, and, finally, what is the exact cause of the error. This implies, then, that the hypothesis points to the possible resolution of the problem. If the analyst finds that the logical scope is too broadly defined to allow him to formulate an adequate hypothesis, he reenters the localization process (see Section 3.3.2) to refine (i.e., narrow) the logical scope and thereby help isolate the error's location and cause. If the analyst is successful in formulating a hypothesis, he tests its validity by simulating, in some manner, the effect of the hypothesis' implementation. Methods that may be employed to test a hypothesis include mental modification of the source program, actual modification of the object program and its environment during or before execution by use of debugging tools, and simulation of the proposed change by use of simulation tools. If the test results indicate that the hypothesis was invalid (i.e., its implementation had no effect on the anomalous condition), the analyst returns to the localization process to refine the logical scope and thereby gain further insight into the error's location and cause. If the test results indicate the anomalous condition is only partially explained, the analyst also returns to localization to refine the logical scope or, if he is convinced there is more than one error, to define a new logical scope in search of another error. Even if the test results indicate the anomalous condition is completely explained, the analyst may not have sufficiently pinpointed the error to be able to recommend a solution and once again he returns to localization to refine the logical scope.

As can be seen from the above description, hypothesis formulation/test may be an iterative process performed at various levels of abstraction. For example, the analyst may first hypothesize that a particular subcomponent is returning an incorrect value to the calling component. If testing substantiates this hypothesis, the analyst refines both the logical scope and the subsequent hypothesis to explain more specifically where within the subcomponent the error occurs and what is its cause. Hypothesis formulation/test may also be an incremental process in which the analyst first attempts to explain part of an error and then, if he is successful, he uses the partial explanation as the basis for formulating a hypothesis that completely explains the error's cause. Only when the analyst feels confident that he fully understands where, when and why the error occurs and what must be done to correct it, does he enter the resolution/termination process (see Section 3.3.5).

Figure 3-7 depicts the steps in the hypothesis formulation/test process. This process is described in greater detail in the subsections that follow.

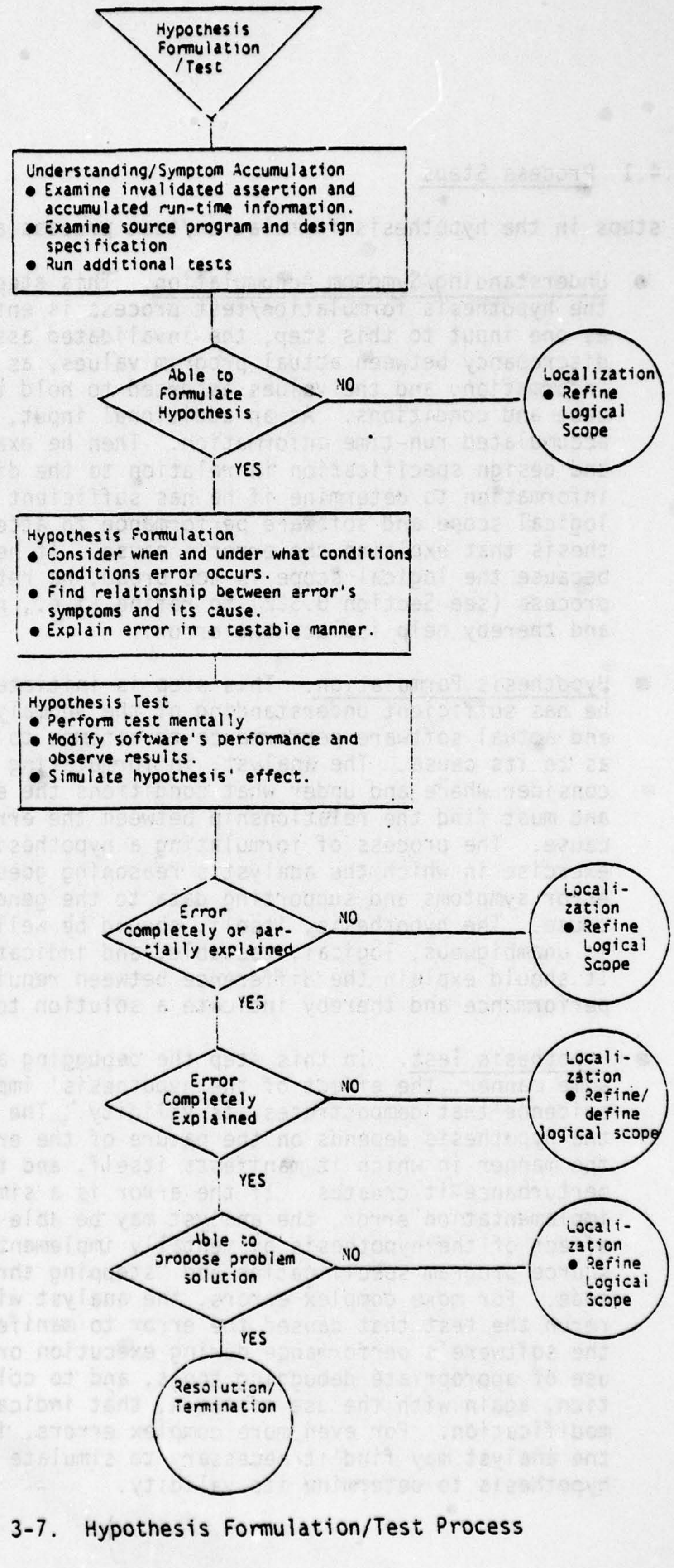


Figure 3-7. Hypothesis Formulation/Test Process

3.3.4.1 Process Steps

The steps in the hypothesis formulation/test process are described below.

- Understanding/Symptom Accumulation. This step is performed each time the hypothesis formulation/test process is entered. The analyst uses, as one input to this step, the invalidated assertion, i.e., the discrepancy between actual program values, as evidenced by run-time information, and the values asserted to hold true for the given test case and conditions. As an additional input, he uses the previously-accumulated run-time information. Then he examines the source program and design specification in relation to the discrepancy and run-time information to determine if he has sufficient understanding of the logical scope and software performance to attempt to formulate a hypothesis that explains the error's cause. If he is unable to do this because the logical scope is too broad, he returns to the localization process (see Section 3.3.2) to refine (i.e., narrow) the logical scope and thereby help isolate the error.

- Hypothesis Formulation. This step is initiated when the analyst feels he has sufficient understanding of the anomaly between the expected and actual software performance to attempt to formulate a hypothesis as to its cause. The analyst, in formulating the hypothesis, must consider where and under what conditions the error manifests itself, and must find the relationship between the error's symptoms and its cause. The process of formulating a hypothesis is an inductive exercise in which the analyst's reasoning goes from the particular error symptoms and supporting data to the general explanation of the cause. The hypothesis, itself, should be well formulated in that it is unambiguous, logical, testable, and indicates predictable results. It should explain the difference between required and actual program performance and thereby indicate a solution to the problem.

- Hypothesis Test. In this step the debugging analyst simulates, in some manner, the effect of the hypothesis' implementation to obtain evidence that demonstrates its validity. The method used to validate the hypothesis depends on the nature of the error and its cause, the manner in which it manifests itself, and the extent of the perturbance it creates. If the error is a simple and very localized implementation error, the analyst may be able to determine the effect of the hypothesis by mentally implementing the change in the source program specification and "stepping through" the resultant code. For more complex errors, the analyst will find it necessary to rerun the test that caused the error to manifest itself, to modify the software's performance during execution or prior to it by the use of appropriate debugging tools, and to collect run-time information, again with the use of tools, that indicates the effect of the modification. For even more complex errors, including design errors, the analyst may find it necessary to simulate the effect of the hypothesis to determine its validity.

After testing the hypothesis, the analyst may find that he has completely explained the anomalous condition being debugged, partially explained it, or completely failed to explain it. In the latter case, the analyst returns to the localization process (see Section 3.3.2) to refine the logical scope and thereby gain further insight into error. If the error is only partially explained, the analyst also returns to localization but his reason for doing so depends on the level of understanding he has achieved. If the analyst feels confident in both the correctness and completeness of the hypothesis and in his ability to recommend a solution based on it, then, most likely, more than one error has caused the anomalous condition. In this case, he reenters localization to define a new logical scope which contains another error. Otherwise, he reenters localization to refine the current logical scope so that he may gain sufficient understanding to formulate a more comprehensive hypothesis. Even if the hypothesis has completely explained the error, the analyst may lack sufficient understanding of it to recommend a solution. In this case, also, the analyst returns to localization to refine the logical scope. Only when the analyst feels confident that the hypothesis completely explains the error and that he is able to recommend a solution, does he enter the resolution/termination process (see Section 3.3.5).

3.3.4.2 Process Relationships

The relationship of the hypothesis formulation/test process to the other debugging processes is shown in Figure 3-7. This process is entered from execution analysis when the analyst has established that the entry assertions associated with the logical scope have been validated but the exit assertions have been invalidated. The successor to this process is either: (1) the localization process for the refinement or redefinition of the logical scope, or (2) the resolution/termination process when the analyst has been able to formulate a hypothesis that fully explains the error's cause and to recommend a solution for the anomalous program behavior.

3.3.4.3 Information Requirements

The following information components are used in the hypothesis formulation/test process:

- Logical Scope. Used in the definition of boundaries in the software within which the location of the error cause has been isolated.

- Assertions, specifically the validated entry assertions and invalidated exit assertions. The information verifies that the selected logical scope contains the error cause. The hypothesis is formulated in an attempt to explain the invalidated assertions.

- Application System Descriptions, specifically the requirements, design and source program specifications associated with the logical scope. This information includes the description of the specific test environment relating to the error under investigation. It is used to formulate and test the hypothesis.

- Run-Time Information, including:

  - Error Symptoms. That information which indicates an erroneous condition. The absence of error symptoms after hypothesis simulation helps to validate the hypothesis and assure that program modifications, used to test the hypothesis, will not cause new errors.

  - Program State Information. This information is available if testing the hypothesis requires running the program. It is compared with predicted results as part of testing the hypothesis.

3.3.4.4 Tool Usage

This process uses tools in each of its steps. In the understanding/symptom accumulation step, the analyst may need run-time information to determine under what conditions the error occurs. There are a number of debugging tools* that gather run-time information and help the analyst determine the program state information response to a given test case. They include the following:

- Post-Mortem Dumps (see Appendix A, Section A.1)
- Snapshot Dumps (see Appendix A, Section A.2)
- Breakpoint/Trap Dumps (see Appendix A, Section A.3)
- Monitor Dumps (see Appendix A, Section A.5)
- Dynamic Internal Trace (see Appendix A, Section A.7)
- Recorded Trace (see Appendix A, Section A.8)
- Monitor Trace (see Appendix A, Section A.9)
- Software Breakpoint/Trap (see Appendix A, Section A.16)
- Hardware Breakpoints (see Appendix A, Section A.17)
- Reversible Execution/Backtracking (see Appendix A, Section A.18)
- Program Execution-Oriented Recording/Reduction (see Appendix A, Section A.22)
- Input/Output-Oriented Recording/Reduction (see Appendix A, Section A.23)

---

* Further discussion of these tools can be found in Section 5 and Appendix A and Section 3.3.2 of Volume III.

In the hypothesis formulation step, the analyst can use a set-use matrix/cross reference analysis tool (see Appendix A, Section A.10) to help determine the variables about which the hypothesis must predict expected values. The analyst first determines which software components are contained within the selected logical scope. He then uses the set-use matrix to determine the entry and exit variables of the software components. The debugging analyst can use a number of tools in the hypothesis test step. If he needs to rerun the test case, he first must reconstruct the test environment (as he did in the verification/duplication process). There are a number of tools available for aiding the reconstruction of the exact test case, and hardware/software configuration including the data base and/or initial conditions. They include:*

- Program Support Library
- Test Input Listing Tool (see Appendix A, Section A.24)
- Script Tape (see Appendix A, Section A.24)
- System Status Summary Tool (see Appendix A, Sections A.26, A.27)
- Checkpoint/Restart (see Appendix A, Section A.28)

Tools that can be used to modify the program in order to test the hypothesis include the following:

- Interactive Modification Tools (see Appendix A, Section A.19)
- Recompilation/Correction (see Appendix A, Section A.20)
- Hardware Modification Tools (see Appendix A, Section A.21)

In this step, the analyst will need the same tools that were identified in the understanding/symptom accumulation step to collect and analyze run-time information that will be compared with predicted results. In addition, he may make use of comparator tools (see Appendix A, Section A.25) to compare program state information generated before and after a program modification was made to determine whether or not the modification caused new errors to occur. Simulators (see Volume III, Section 2.1.2) can be used to test a hypothesis that poses a design flaw as the cause of an error.

3.3.4.5 Assumptions Made

This process assumes that the localization and execution analysis processes have determined a logical scope for which entry assertions were validated and exit assertions were invalidated, and which therefore bounds the cause of the error. In addition, it is assumed that these assertions were correct and complete. If they were not, then the analyst has not, in fact, isolated the cause of error and a hypothesis can not be formulated.

* Further discussion of these tools can be found in Section 5 and Appendix A, and Section 3.4.3 of Volume III.

3.3.4.6 Error Sources/Problem

A primary problem in this process is that the logical scope has not been narrowed sufficiently to enable the debugging analyst to formulate a hypothesis that really explains the invalidated assertions which is assumed to be the error he is debugging. It may not be until a test of the hypothesis has taken place that the analyst will discover that he has attempted to formulate a hypothesis on the basis of insufficient information. Even if the hypothesis may theoretically explain the error cause, not all hypotheses are ammenable to complete validation by simulated implementation. For example, the hypothesis may be that a coded program component deviates vastly from its design. Inspection of the code in light of the design may validate the hypothesis but only the recoding of the component, which is beyond the scope of the debugging methodology (see Section 3.3.5), can validate that the error being debugged will be solved as a result. Similarly, the hypothesis may be that there is a discrepancy in the design in that the values output by one component are inconsistent with the values expected by a calling component. While inspection of the design of the two components may validate the hypothesis, it may be beyond the analyst's responsibility and/or capability to resolve which component's design is in error. In such cases, the analyst, after validating the hypothesis to the extent possible, enters resolution/termination (see Section 3.3.5) under the assumption that he has fully explained the error. Only after the error's solution has been implemented will it be possible to substantiate the validity of the hypothesis. If, at that time, it is found that the hypothesis does not fully explain the anomalous condition, it will be necessary to reinitiate the debugging process to complete the problem resolution.

One outcome of the hypothesis test process is that the analyst may find that he has explained an error but that it is unrelated to the error he is debugging. In this case, he reports the error in an SDR, enters the resolution/termination process to resolve the error he has explained, and restarts the debugging process in search of the cause of the original error.

3.3.5 Resolution/Termination

The resolution/termination process is the last activity addressed in the debugging model. When this process is entered, one of three conditions exists with respect to the error reported in the Software Discrepancy Report:

- Error has not been duplicated or has not been isolated within available/reasonable resources.
- Error has been attributed to a man/machine error.
- Error has been attributed to the software and a hypothesis has been formulated which explains the discrepancy between actual and required software performance.

The purpose of this process is to initiate the action, if any, which will resolve the reported problem.

If the analyst has been unable to duplicate or isolate the error after a reasonable expenditure of resources, as viewed in light of the seriousness of the error, a management decision is needed to resolve the problem. Intermittent software or hardware errors may be so difficult to duplicate or isolate that large amounts of resources are required. Often an error of this type is left open for some period of time in hopes that the error will not manifest itself again, or will be resolved at a later time when additional, and perhaps related, run-time information has been generated in the regular course of testing. It is to be noted that the difficulty of solving intermittent errors places greater demands on control and visibility into the testing/debugging process. Resolution of intermittent problems and, indeed all problems, is seen to be more efficiently achieved when each test is clearly identified, including an unambiguous statement of the cause-effect relationship of the test inputs and test results, and when the test process is structured so as to test increments in software functional capability.

If the analyst has determined that the observed and reported problem is due to an error in man/machine interfaces or a misinterpretation of required software performance rather than an error in the software specifications, he resolves the problem by identifying its source and offering evidence to substantiate his conclusion. If the error has been caused by inadequate/ inaccurate documentation, including software specifications, test procedures, and user's instructions regarding input data preparation, initial conditions, limitations/restrictions, and hardware/software configuration, the analyst would be wise to recommend upgrading the appropriate documentation as part of the problem resolution.

If the analyst has determined that the reported problem is an error in one or more software specifications, he must document his findings, including a description of the error's cause, the conditions under which it occurs, and the proof that his hypothesis adequately explains the difference between specified and actual software performance in terms of the given implementation. The analyst must also identify the actions that are necessary for problem resolution, as indicated by the hypothesis. The required action may entail modification to one or more of the hierarchy of software specifications (i.e., requirements, design, and source language specification); it may also entail retesting and regression testing as well as modifications to the test plan and/or procedures, user's manuals, and other supporting documentation. Whenever possible, the analyst should suggest a work-around procedure, i.e., a temporary fix which allows the tester to continue his activities, circumventing the anomalous condition in order not to impede development progress. It must be recognized, however, that such a procedure is only a temporary solution and that will be necessary for the software to conform to its functional/performance requirements as a condition for acceptance.

Neither the decision to implement the resolution indicated by the hypothesis, nor the actual implementation rests with the debugging analyst as modeled by this methodology. Unless the error is a trivial implementation error, additional analysis may be required to completely specify the solution. This is due to the fact that the analyst's hypothesis was based on an invalidated assertion derived from analysis of system performance in response to given test inputs and conditions rather than the total range of inputs

and conditions that must be handled by the software. Additional analysis may be required to ensure that the envisioned solution, as indicated by the hypothesis, will not introduce other errors. Additional analysis may also be required to evaluate the impact of the proposed solution on the cost, schedule, and performance constraints of the development effort and may require that alternative solutions be sought. None of these activities are debugging activities.

As can be seen from the above description, the resolution/termination process is a transition step between debugging and the other software development activities that proceded it. It was in those activities that the error was introduced, and it is the responsibility of those activities to resolve it once it has been isolated by the debugging process.

Figure 3-8 depicts the steps in the resolution/termination process.

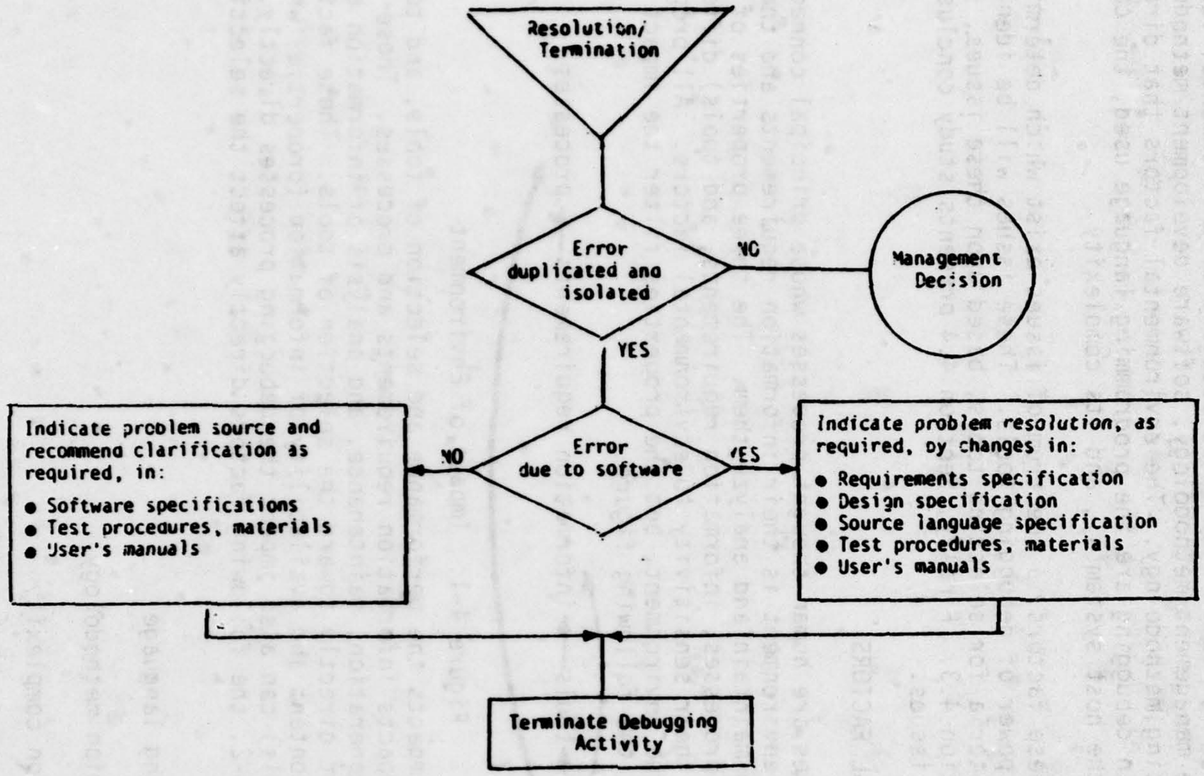


Figure 3-8. Resolution/Termination Process

4. ENVIRONMENTAL IMPACT ON THE DEBUGGING METHODOLOGY

This section examines the factors that determine the environment within which debugging is performed. These factors are identified in Section 4.1 and are divided into two categories: (1) factors that influence the quality of the application system descriptions, and (2) factors which influence the selection of debugging tools. The first category of environmental factors includes software management methodology, software development methodology, and software testing methodology. The environmental factors that directly impact tool use in debugging are the programming language used, the computer architecture of the host system(s), and its complexity.

Within each of these factors, a spectrum of issues exist which determine the availability and power of debugging tools. These issues will be identified in Section 4.2. Criteria for selecting tools, based upon these issues, is developed in Section 4.3. Finally, Section 4.4 presents study conclusions based upon these issues.

4.1 ENVIRONMENTAL FACTORS

Debugging processes are human thought processes whose principal connection to the debugging environment is their information requirements and the tools used to generate, maintain and analyze them. The three properties of the debugging model (processes, information requirements, and tools) differ significantly in their sensitivity to environmental factors. All properties are impacted by the environment, but the properties filter the impact of the environment as in the following figure.

$$\begin{tikzcd}\text{Environment} \arrow[r] \arrow[rr, bend left] \arrow[rrr, bend right] & \text{tools} \arrow[r] & \text{information requirements} \arrow[r] & \text{processes}\end{tikzcd}$$

Figure 4-1. Impact of Environment

The environment impacts the performance and selection of tools, and to a lesser extent, impacts information requirements and processes. Those factors that affect the generation, maintenance, and analysis of information are those applied most directly toward the selection of tools. Those factors that affect the content and availability of information (properties which may not depend on tools) can also impact the debugging processes directly. As shown in Figure 4-2, the following factors directly affect the selection of debugging tools:

- Programming language
- Verification methodology
- Application complexity
- Computer system architecture.

	Properties of Debugging Methodology					
		Information Requirements				
		System Descriptions		Run-Time Information		
Debugging Environmental Factors	Requirement Specification (Section 4.2.1.1)	Design Specification (Section 4.2.1.2)	Source Language Specification (Section 4.2.1.3)	Error Symptoms (Section 4.2.2.1)	Program State Information A (Section 4.2.2.2)	Tools (Sect. 4.3)
Software Management Methodology (Section 4.1.1)	Maintenance	Maintenance	Maintenance	Maintenance		Selection
Software Development Methodology (Section 4.1.2)	Quality*/ Representation	Quality*/ Representation	Quality*/ Representation			
Verification Methodology (Section 4.1.3)				Generation	Generation	Use
Programming Language (Section 4.1.4)			Content		Representation	Selection
Computer System Architecture (Section 4.1.5)					Content	Selection
Application Complexity (Section 4.1.6)				Content	Content	Selection

* Related primarily to the results of human thought processes.

Figure 4-2. Impact of Environment on Debugging Methodology

Application system descriptions are impacted more directly by environmental factors that affect their maintenance and quality (software management and development methodology) than by the tools used to generate and analyze their content. The content, generation, availability and representation of run-time information are more directly affected by the factors that impact the selection of tools used to generate and analyze the data than by other environmental factors.

The debugging processes are principally synthetic human thought processes, based upon conceptually integrating the symptoms exhibited by the system's actual performance with the descriptions of the system's required performance. This builds a theory of the system's expected behavior that leads to an explanation for the differences between specified and actual performance. The steps within the debugging processes which are information gathering steps are directly affected by the availability, content and format of the information that is examined. In the following sections, specific environmental factors are examined for their impact upon debugging information requirements.

4.1.1 Software Management Methodology

Software management methodology can improve the debugging process by establishing, promulgating and enforcing the use of a formal structure for the conduct of testing. The techniques utilize configuration and technical control procedures to maintain the integrity of the environment and the application system descriptions. These procedures (1) identify and document the physical characteristics of configuration items, (2) control changes to those characteristics, and (3) record and report the status of change processing and implementation. It is management's task to ensure that these procedures are carried out. Because the procedures are essentially clerical activities, they can be established as purely manual procedures. A program support library (PSL), which is a set of manual or automated clerical procedures to control the source and object code of all software to be tested, the test cases and test results can support already established procedures. A PSL, without management enforced procedures, can do little in maintaining the integrity of software configuration.

The debugging analyst's success, in the reconstruction of the test environment (i.e., test, software and hardware configuration), within which the system failure occurred depends upon configuration control procedures being rigorously carried out. The procedures help the debugging analyst determine the exact version and modification identifiers of all system, computer, and hardware components that were involved; test procedures that were being performed; and the correspondence between the application system descriptions and the computer programs under test and the status of changes to each. In addition, the organized management of testing and debugging requires that a document such as a software discrepancy report be used to initiate the process of finding and correcting the error. The software discrepancy report links the description of the observed error with an identification of the test and hardware and software configuration, which together define the test environment which demonstrated the problem. It is especially important to document software errors when the testing responsibility is separate from the debugging responsibility.

4.1.2 Software Development Methodology

In terms of the information requirements of the debugging methodology, the software development methodology establishes a standard for the content of the application system description. Ideally, the system description documents (requirements, design, and program source specifications) will exhibit the desirable attributes of good structure, traceability and minimum complexity to achieve required goals. Various methodologies have been proposed which aim to help achieve these goals throughout the life cycle. They utilize such techniques as hierarchical decomposition, functional analysis, HIPO and SADT. Programming languages have evolved to support these methodologies, and existing languages have been modified or preprocessors have evolved to support the same end. These issues and concepts have all been widely addressed in the literature, and will not be explored here. Volume III of this report examines software development methodologies at some length.

4.1.3 Verification Methodology

Verification methodology in the context used in this study includes the verificiation processes, including testing, that immediately precedes debugging. The verification methodology can impact the debugging process in several ways:

- Potential problems are found and eliminated before the integration testing phase. The whole thrust of current software engineering practices is to reduce the number of logical errors introduced at each phase of the development process in order to eliminate errors as early as possible. Verification is the process of determining whether the representations of selected levels produced during the CPCI development process fulfill the requirements levied by the higher levels. Given that the application system descriptions represent transformations of the system at specific levels of abstraction, verification is concerned with eliminating logical flaws between the transformations, as well as within each transformation.

- The testing environment is controlled so that the conditions that existed when the system failure occurred can be reconstructed.* It is crucial for both the testing and debugging processes that the environmental elements (test procedures, hardware and software) be identified and controlled so that the testing process discovers the problems, and does not create them. Debugging can be exceedingly more difficult if the test environment is not configuration controlled as well as technically controlled.

- The physical and procedural environment in which the debugging analyst works can facilitate the debugging process. See Section 3.1.4 of Volume III for a discussion of the potential of interactive assistance to the debugging processes.

---

* See Section 3.4.3 of Volume III for a description of methods to reconstruct the test environments.

4.1.4 Programming Language

The selection of a programming language impacts the debugging process in the following ways:*

- Language constructs can enforce the practices of modern software engineering concepts and thereby reduce the number of logical errors introduced into the source language programs.
- The capacity of an associated compiler to detect static errors in the source language program at compile time is dependent on the levels of abstraction supported and the amount of redundancy (e.g., cross checking of data types and data uses) required in the expression of a program in source language.
- The debugging analyst's task of examining the source language programs for logical flaws is simplified if the programming language, as well as the development methodology, encourage the implementation of a program that was adequately and clearly formatted and was written in accordance with structured programming principles.
- The representation of program state information in a form which is easily understandable and readable to the debugging analyst depends upon features of the programming language, the tools used to translate the source language program to an executable program, and the tools used to report the machine state information.

The programming language and its associated compiler is the most important component of the abstract machine** seen by the programmer. As such, the impact of programming language on the debugging process is crucial and is discussed in detail in this section.

---

* Section 3.1.3, Programming Language Considerations, of Volume III covers this topic in greater detail.

** Section 2.2 of this volume defined the concept of abstract machine.

4.1.4.1 Language Constructs

Programming languages can support the programmer in utilizing the principles of levels of abstraction, modularity and structured programming. The capability of a programming language to provide abstract means of describing data and operations on that data can result in the detection of many errors during the design and code activities. These abstractions support the programmer's logical view of his problem which simplify the stepwise refinement of design specifications to the level of source language representation.

4.1.4.2 Error Detection

Programming languages provide a capability to detect source language program errors which otherwise could escape detection until the integration phase. The detection of illegal program constructs requires special language features. Two types of error detection techniques can be used by a language compiler: predictive and confining. A predictive technique allows a property of a program to be determined without reference to the exact data upon which it will operate. These static checks are commonly performed at compile time. Examples of predictive error detection techniques include strong type checking, compile time checking of formal versus actual parameter lists, enforcement of variable scope rules, and built-in synchronization primitives for abstract data types. A confinement technique prevents a program from employing a machine in such a way that the machine does not legally implement the language. Using a confinement technique for a part of a system guarantees either that the part will always function as specified or that a malfunction will be detected. (This technique would not necessarily be applicable to an entire system because of possible run-time constraints). Confining techniques are usually implemented as dynamic checks which are performed at run-time. Modern run-time confinement techniques, such as array bounds checks and capability-based protection systems, may not surface a problem until integration testing because of the large number of possible inputs. They will, however, suspend operation at the program state in which the error occurred, thereby preventing delayed recognition of the problem and destruction of error symptoms. This technique greatly aids debugging by halting execution at the point of the error so that iterative localization is not necessary.

4.1.4.3 Source Language Representation

The source language specifications represent the final transformation in the conversion of required capability into an executable implementation. The readability and reliability of the representation is greatly influenced by the suitability of the programming language chosen for implementing the application problem. The quality of the source language representation, from the debugging analyst's point of view, is highly dependent upon the programmer's use of the source language features and his adherance to effective coding standards and conventions. Applicable techniques which help improve source language representation include:

AD-A069 539 SYSTEM DEVELOPMENT CORP SANTA MONICA CALIF F/G 9/2
SOFTWARE DEBUGGING METHODOLOGY. VOLUME I.(U)
APR 79 M FINFER, J FELLOWS, D CASEY F30602-77-C-0165
UNCLASSIFIED RADC-TR-79-57-VOL-1 NL

2 OF 2
AD
A069539

END
DATE
FILMED
7-79
DDC

- Standards and conventions documents to provide guidelines on coding practices.

- Structured programming constructs to reduce complexity in programs.

- Programming language preprocessors to overlay structured programming constructs on a language (e.g., FORTRAN) which may be deficient in structured language constructs so that it can be better used in structured programming.

- Problem-oriented languages are used for specific problem types by adopting specific phrases and keywords related to the problem from the standard vocabulary (e.g., HAL/S is especially suited for space and aircraft applications).

4.1.4.4 Programming Language and Program State Information Representation

It is especially useful to the debugging analyst to have the program state information representation in the same terms as the source language representation, but often names with less content are used. The program state information is the bridge between the actual software performance on the abstract machine and the reported system performance which may be incomplete and symptomatic of actual system performance. To the extent that program statement information is reported in terms of system description (e.g., source language specification), it can be used to relate the required system performance with actual system performance. This is affected by the capability of the compiler to maintain data generated during the translation of a source language program to an object language program, and the existence of debugging tools that can use the data. The impact of programming language on the selection of tools is discussed in Section 4.3.1.

4.1.5 Computer System Architecture

The architecture of a computer system was defined by G. Amdahl in 1964 as "the attributes of a system as seen by a programmer, i.e., the conceptual structure and operational behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation." ** This definition is still valid today. This concept of computer architecture is readily extended to include system software provided by hardware vendors. This study has extended this concept to include the total hardware and software configuration (with corresponding documentation) used by the programmer and has called the totality an abstract machine. The importance of a programming language and its associated compiler in the debugging process was discussed in the previous section (4.1.4). This section is concerned with the properties of the physical component of the abstract machine. This study initially concentrated on a breakdown of computer system architectures into large scale systems, minicomputers, microcomputers, and networks. Within the first three classes of architectures, it was found that debugging environmental issues derived from computer system architecture varied as much within a classification (i.e., between the IBM 370

** Amdahl, G.M., Blaaun, G.A., Brooks, F.P., "Architecture of the IBM System 360". IBM Journal of Research and Development, Apr. 1964.

and the Honeywell 6180) as they did between classifications (i.e., large scale vs. minis).

Certainly it is true that the architectural view supported by the instruction set processor, memory addressing schemes, and basic operating system features constitute an important part of the environment in which debugging is performed. The study team was not able to identify generic attributes of the type that correspond to the large scale, mini, micro classification of computer system architectures. The primary issues that were identified as computer architecture related are the amount of resources available (especially execution speed, memory size supported, and I/O device mix) and the vendor-supplied software tools.

The rapid pace of technical development by computer manufacturers has blurred the physical performance margins between large scale computers and minicomputers; and between minicomputers and microcomputers. The level of support provided by system software of various size hardware systems greatly impacts the debugging process. There is an estimated 10-year gap in the level of sophistication between the software support offered by manufacturers of large scale machines and that offered for minicomputers. A similar gap separates mini and microcomputers. The reasons for these gaps are due to both economic and technical factors, as discussed below:

- Vendors of large scale machines have had a longer period of time to evolve a dialogue with and meet the needs of their user communities. The users of large machines expect and receive a high degree of vendor support. Vendor investment in sophisticated system software, including diagnostic features for compilers and debugging tools, is substantial and not easily duplicated by mini and microcomputer vendors.

- Debugging tools and compilers generate system overhead which can normally be tolerated in a large multiprogrammed environment. This overhead may not be acceptable in a small machine environment, due to timing or memory constraints.

- Users of minicomputers tend to be technically sophisticated and willing to develop their own extensions to vendor-supplied software. The lower cost of minicomputers allows programmers to operate their programs in a hands-on environment. Since minicomputers have begun to approach large scale machines in computing power, market forces have begun to support high level system software features for minicomputers.

4.1.5.1 Large Scale Machines

Due to their longer period of development history and to the size of their resources, large scale machines generally offer the most powerful debugging tools available. These tools require substantial investments of time, effort, and money for both development and proficiency in use. They generally induce inefficiencies in utilization of time and storage space, but these limitations have minimum impact on large systems. Although many military

systems, which are hosted by large machines, have resource constraints, these machines can generally support the inefficiencies of powerful debugging tools during program development.

4.1.5.2 Minicomputers

The range of debugging tool capability is limited for minicomputer users by economic and resource issues. The tools which are available tend more towards the traditional set of machine level dumps, traces, and breakpoints.

4.1.5.3 Microcomputers

Self-resident debugging tools for microcomputers are limited to primitive store inspection capabilities. Microcomputer program development systems have developed test and debugging techniques in which a separate processor (micro or mini) is used to provide monitored data of microprogram operation. Extensive use is also made of interpretive computer simulation in which the execution of a program under development is simulated on a large scale machine. This technique allows collection of a wide variety of debugging information.

4.1.6 Application Complexity

The last environmental factor for debugging is the complexity of the application area. This area includes the complexity of the overall computer system architecture, as in networks of computers, described in Section 4.1.5. It also addresses the complexity inherent in the system goals, as in artificial intelligence applications. The number of components involved in realizing system goals, and the richness of their interaction, varies not only with the quality of the design, but also with the difficulty of the problem. A small numerical calculation is intrinsically less complex than a real-time application involving concurrent processes. This study could not begin to consider the impact arising from all possible application areas because of their great number and variety. The classification of application areas into categories that impact debugging in a specific manner would be a useful effort. Several issues that arise from application complexity that were addressed in this study relate to applications that utilized networks of computers and exhibited concurrent rather than sequential processing.

A program application which distributes system functions across multiprocessors or among a network of computers is a unique factor that has serious effects on debugging over the entire computer architectural spectrum. In this application, two or more sequential processes are executing during the same time span while cooperating among themselves to achieve system objectives. Such configurations are common on multiprocessor computers and in cooperating networks of computers. In general, no assumptions can be made about the relative speed of the processes involved. The cooperation between processes is achieved through some form of shared data. In addition to the ordinary concerns for correctness of the individual processes, such systems historically have been prone to several forms of time dependent errors to which purely sequential programs are not subject. These include race conditions, hazards, and deadlocks.

While the behavior of a sequential program is uniquely determined by the value of its input variables, the behavior of a multiprocessor system is influenced by the order in which its processes execute their operations. Thus in determining the validity of the design of such a system, the synchronization of the processes must be considered as well as their functional integrity. This implies that the element of time must be included in the program state. The amount of complexity introduced by these considerations is profound. A measure of this fact is that a theory of proving sequential programs correct is becoming established, and, while not currently of practical use for large systems, this theory is providing new insight into testing and debugging sequential programs. The theoretical study of verifying networked systems has not produced even these limited results, due to the added complexity which is faced.

In a system environment with a single processor, the program state information may appear to the debugging analyst as a program history in two dimensions - that of sequential operation and name-space values. In multiprocessing, the program's execution history is not given in terms of just these two dimensions for each individual processor, but the program state information must also provide the computational history of the program in terms of correspondence of time and shared data between processors. That is, if the debugging analyst must examine name-space values and the location of termination for a program operating in one processor in a network, he must also determine the location and values of shared data at the same instance of time for the programs that are operating in the other processors.

4.2 IMPACT OF ENVIRONMENTAL FACTORS ON TOOL USE

This section discusses the role of tools in supporting the debugging methodology and the manner in which environmental factors affect the use of tools. Information requirements of debugging are satisfied by a combination of manual and automatic means. Each of the environmental factors identified in the previous section raises issues which determine the extent to which automated tools can support debugging. These issues are explored below.

4.2.1 System Description Maintenance Tools

The system description information components are the requirements, design and source language specifications, and the error report itself. Each of these components depends primarily upon the human environmental factors of software management and development methodologies for their integrity. Due to the large amount of information involved, and the tedious and error prone processes used in maintaining these information components, a number of automated tools have been developed to aid in these processes. These tools are, in general, not strongly impacted by the technical environment factors, since they need not be hosted by the development system.

It should be mentioned that the fundamental qualitative measure of the system description information components is the clarity of structure and organization of these documents. The tools to be mentioned in this section can contribute to the establishment of an environment in which this goal is more likely to be met. However, the quality of these documents depends directly upon the talent of their authors.

4.2.1.1 Requirement Specification Tools

A great deal of attention has been directed to automated requirement data base systems which can generate, update, and perform limited consistency checks for requirements specifications. These include CADSAT and SREM. These tools are general enough to support various software management and development methodologies. CADSAT is discussed in greater detail in Volume II, Section 3.2.1, of this report.

4.2.1.2 Design Specification Tools

Since the demarcation of requirements and design specifications is often vague, the above mentioned tools may be viewed as supporting top level design. More detailed design specifications may be produced using various program design notations, some of which can provide automated processing to produce cross reference listings and summary information.

4.2.1.3 Source Language Specifications

This discussion only addresses tool capabilities not provided by a compiler. Various forms of language preprocessors can enforce standards, check assertions, and provide structured control mechanisms. Perhaps more

importantly, the program support library (PSL) tool can substantially aid software management methodology by providing version control for source programs, object load, or test modules. In addition, the PSL can aid reconfiguration of hardware/software/test conditions upon which error reports are based.

4.2.2 Run-Time Information Gathering Tools

Many of the debugging methodology processes utilize and create abstract bodies of information which are models of human understanding and problem solving factors. These types of information, such as assertions, hypotheses, and logical scope are generally created by a human thought process; there are very few tools which assist in their creation. The more concrete information components of the debugging methodology, actual program behavior (error symptoms) and program state information, are collected by tools to support the creation and validation of the more abstract information entities.

These tools provide information within a context determined by the environmental factors of verification methodology used, programming language used, computer architecture of the host system, and the complexity of the application. Tools which gather program state information are primarily variations and combinations of two basic kinds of information:

- Values of program variables.
- Identity of current (abstract) machine operation, and historical sequences of such operations.

Appendix A of this volume contains an extensive description of tools incorporating these capabilities.

4.2.2.1 Error Symptom Information

Since error symptom information is defined in terms of the input/output relationships exhibited by a program, tools which provide this information are largely application specific. Many vendor operating systems provide limited data of this nature in the form of audit trails on file usage, a day log of all operating system directives, etc. If the program deals with large files of data as inputs and outputs, manual verification of the file integrity and correctness is a tedious and error prone process. Application specific tools for checking file consistency and content are a tremendous aid in debugging. Also of great use are independently programmed programs that can verify input/output relationships of a program, hopefully based upon a different algorithm or acceptance test than used in the program. For example, a program that utilizes sophisticated numerical analysis routines to compute orbits for satellites can be checked with a program that compares the results with certain physical constraints on the solution.

4.2.2.2 Program State Information

The thought processes of the debugging model require program state information in terms of the variables and operations of the source program. This information can be directly supplied by symbolic debugging tools or it may have to be generated by a laborious manual process of reconstructing symbolic values from physical machine level program state data. Symbolic debugging tools require close interaction with the compiler for the language being used. Their availability is largely a matter of vendor support, as they are usually considered too complex for on-site development.

Practically all vendors provide basic dump/trace tools which depict program states in terms of a particular computer architecture. More advanced symbolic tools will require more of the hardware's resources, and may prove unusable for programs which approach system resource limits. As a result of these factors, symbolic debugging tool availability is strongly dependent on both programming language and computer system architecture environmental issues.

The complexity of the application also impacts the selection and availability of program state debugging tools. Because this issue is intrinsically application dependent, debugging tools used to cope with complexity are also application dependent. Computer networks may require customized hardware to collect system wide states, even at the basic machine levels. Special software is often written to provide monitoring of message traffic between individual computers in a network. For complex applications on a single computer, it is often desirable to include tools in the system design which provide formatted dumps of program data abstractions.

4.3 Tool Selection Criteria

Within the constraints defined by debugging information requirements and environmental factors, it is necessary to develop a set of criteria by which debugging tools can be selected for use in a project. These criteria will be modified based upon experience in the specific applications involved. This section develops an initial set of tool selection criteria which are sufficiently general to apply to any program development activity. The criteria are organized according to the information structures needed by the tool-using processes as different processes may use the same set of tools to meet their information requirements.

4.3.1 Collection of Program State Information

Program state information is the key information component which is collected by software and hardware tools. It is the primary product of the vast majority of debugging tool systems. The single most important criteria for judging program state collection tools is whether the tool can exhibit the dynamic state behavior of a program in terms of the level of abstraction with which the debugging analyst is concerned.

In any programming environment, the necessary information for reconstruction of the state of a program's data structures is available in some form. At the most primitive level, the contents of hardware registers and memory words can be viewed one at a time on the front panel of a computer. At the opposite extreme, sophisticated interactive debugging packages exist for high level languages such as PL/I which allow selective tracing and dumping of program statements in terms of variable names, source statement number, procedure entries and exits, etc. These tools may even allow source level breakpoints and source modification before execution continues. As discussed elsewhere, an even higher level of abstract state information is obtainable by use of programmer supplied debugging procedures.

Within this range of capability to display program state information, an individual debugging analyst will be provided with a fixed set of available tools. The analyst must then select which tools to use based upon criteria such as the following:

- Compatibility with run-time environments
- Resource utilization of tool
- Analyst familiarity with tool use
- Requirement for recompilation
- Time required for use
- Availability of interactive terminal
- Attended/non-attended computer time.

4.3.2 Detection of Error Symptoms

In the process of debugging a problem, it is necessary to be able to decide if a specific program response to specific program input data satisfies the requirements for program behavior. It is often necessary to determine the program's response to conditions not present in the test conditions upon which the problem report was based. For systems which have large volumes of input and output data, it is useful to have access to test data generating tools and program output analysis tools. These tools are application specific and must be developed as part of the system development effort. Some research has been directed at automated generation of test case data from source text, but results to date are limited to single-module-sized programs.

4.3.3 Maintenance of Application System Description

The verification/duplication process relies primarily on procedures which control system configuration. The criteria by which such procedures should be judged include the extent to which they provide the following capabilities:

- To identify the specific test cases (data and procedures) which were used

- To identify exactly what collection of software modules was linked together to form the load module which was tested
- To identify exactly the hardware environment of the test run
- To identify exactly the system environment (files, operating system) of the test run
- To identify the approved changes that were made to the requirements, design and source language specifications.

The program support library (PSL) provides substantial support for the procedures that create a test environment in which these criteria may be met. The PSL provides a controlled environment for the control of machine processible elements such as software modules and test cases. Configuration control procedures should be chosen that satisfy the last criteria for requirements and design specifications.

4.4 CONCLUSIONS

There are several conclusions which can be drawn via this examination of environmental impact on debugging information requirements. Figures 4-1 and 4-2 summarize many of the aspects of this relationship. The following sections present specific conclusions concerning these issues.

4.4.1 Filter Effect of Methodology

Software management methodologies go hand in hand with software development methodologies in creating an environment in which most errors simply do not survive to appear during system integration. Modern design methodologies can attempt to ensure that those errors which do appear during integration are confined in impact to small portions of the program.

4.4.2 Filter Effect of Languages

Many of the errors which have historically been discovered during integration are not interface errors. They simply represent test cases not previously encountered. Many of the error confining and predicting techniques of such languages as Algol 68 and Pascal are capable of reducing the incidence of such errors.

4.4.3 Planning for Application Specific Tools

Debugging tools included as an integral part of a system design provide the most powerful capabilities of all tools. They should be included as part of the basic design approach whenever the application complexity becomes large.

4.4.4 Planning for Debugging Tool Resource Requirements

Much more productive use will be made of analysts' time if high level language oriented debugging tools can be used. When planning the resource requirements of a system, the resource overhead imposed by such tools should be considered.

5. A MODEL DEBUGGING SYSTEM

A model debugging system is a software support facility which provides the debugging analyst with the capabilities he requires to isolate and resolve a software error in terms of the environment he is using and is familiar with. This section addresses procurement, design and implementation aspects of such a debugging software support system.* It is presented to define an environment which supports the use of the debugging methodology presented in Section 3. In addition, it deals with some of the major problem areas found in debugging capabilities offered in existing software support systems, including:

- Inefficient allocation and use of resources due to non-use and/or duplication of debugging software capabilities;
- Lack of early planning for debugging activities, leading to ad hoc debugging software support which does not necessarily meet the needs of debugging activities;
- Lack of a clearly defined development methodology, especially with regard to testing/debugging, making it difficult for the Air Force software development engineer (as well as software development manager) to determine the adequacy and efficiency of software tool support for debugging activities for the system at hand.

These problems are software development management problems, and appear to reflect a general lack of understanding of the activities in the testing/debugging process. Because of this, this section is intended to briefly discuss related issues in an attempt to provide management insight into the procurement, design, and implementation of debugging software support capabilities. It is intended to be applicable to the development or up-grade of debugging software support capabilities for embedded software procurements. The information provided will better enable the Air Force software development engineer, or equivalent, to evaluate debugging environments either deemed necessary, proposed, or provided by software development contractors.

Little attention is currently being focused on debugging capabilities used or offered by software development contractors in embedded software system procurements. Debugging capabilities are more or less assumed to exist as either part of the specific development contractor's support software, or part of the vendor-supplied hardware/software. However, the results of this study emphasize the need for a well engineered, comprehensive software development methodology which provides the information structures needed during integration-level testing/debugging; and ideally, the debugging software support system should provide run-time information in a syntax and format understandable to the debugging analyst in terms of his specific environment. That requires significantly more attention focused on the interrelationship of the software development methodology and the debugging software support system used by the development contractor than has generally been given.

---

* Appendix A describes features of generic debugging tools that support their use in generating, analyzing or maintaining the information needed by the debugging process model.

Even in procurements where independent verification and validation (IVV) contractors are used, the software development methodology claimed and the software support system used is generally not integrated in such a manner so as to fully offer a "model" debugging system.

While debugging as a separate function of the software development process may not be receiving great attention, there is a widely based, informed opinion in the software engineering community that coordinated use of selected automated tools and manual methods, both based on and integrated with a consistent software development methodology, are necessary to significantly improve the capability to produce and maintain reliable software. The application of software engineering techniques to the various and diverse functions and sub-functions of the software development process are intended to achieve the integration of those functions in a meaningful and controlled manner. Ad hoc application of the tools and techniques will not produce the same level of technical control and management visibility of the product as does the well-disciplined software engineering approach.

Definition of some of the characteristics of a model debugging system presented in this section is a step toward the development of a more fully integrated software development facility. Such a facility can provide central support for an integrated collection of tools and methodology, appropriate for all phases of the software development process, from requirements definition through maintenance and enhancement. This facility, referred to as a software engineering facility, includes support for requirements and design analysis and specification, implementation, verification and testing, configuration management, and overall project management. Such facilities are intended to support the development of reliable and maintainable software. A software engineering facility is built around:

- A data base that contains information about the software system being developed (i.e., representations of system modules, properties of the representation, structural relations between the modules and status of development);
- A command language/processor that provides effective interaction between the user and the facility and maintains the integrity of the data base;
- An open-ended set of software tools that provide the required software development support;
- A comprehensive software development methodology that unifies and structures the use of the data base and automated components.

While this study is primarily concerned with the concept of a model debugging system, examination of the information components needed during debugging indicate that most activities of the development process are implicated during the process of debugging. It can be seen, then, that a model debugging system can only be implemented within the context of the total systems approach for software development, i.e., the software engineering facility. The other components of such a facility (i.e., the data base, its associated data

management system, a command language/processor) provide the mechanisms for control of and visibility into the software development process, including the testing/debugging activities. The cost of developing an integrated software engineering facility is high, generally too high for a single development project to bear itself. The use of a software engineering facility, such as FASP (Facility for Automated Software Production) currently in operation at the Naval Air Development Center, is one attempt to offer an integrated support system to numerous development efforts and thus share costs. The lack of a consistent development methodology and using policy/procedures are seen to inhibit its overall effectivity, but it has been quite a successful attempt to provide software support to the development process. Because the cost of development of a software engineering facility is high, a facility shared by numerous projects, such as the National Software Works (NSW), appears to be the only reasonable economic approach to offer an integrated capability to numerous user communities.

The following sections suggest steps needed to define, develop and use a model debugging system. Again, while this system is addressed as a standalone capability, and as such it can improve the efficiency of the debugging process, it requires that the entire development process be integrated to be truly effective.

5.1 ISSUES RELATED TO PROCURING A MODEL DEBUGGING SYSTEM

This section addresses some of the issues relating to the acquisition of a debugging software support system which will be applicable to the methodology described in Section 3. While an integrated software engineering facility is beyond the scope of this study, it is seen to offer the environment which best supports debugging activities. Such a facility is not addressed in the following sections.

5.1.1 Assessment of Need

Embedded software procurements proceed through a lengthy analysis period in which cost, schedule and performance trade-offs are assessed in relation to the operational capability required. Since the cost of software continues to become an ever-increasing percentage of the total system cost, the life cycle costs of developing and maintaining application software is an issue of much concern. This study has concentrated on debugging during integration-level testing, but maintenance of software necessarily implies changing and correcting operational software, which also requires testing and debugging. Since the costs of software maintenance are staggeringly high, a debugging support capability should not be procured or implemented by the software development contractor for just his integration testing/debugging activities, but should be procured, delivered and accepted by the Air Force to facilitate the enormous burden of maintaining an embedded software system.

The higher the need for reliable software, the higher the need for an integrated debugging support system. However, the costs of providing such a debugging system should be commensurable to the reliability requirements of the software. Procurements requiring a high degree of reliability, such as security or manned-space flight applications, or services offered by an IVV contractor, require a well-defined and disciplined, software engineered

development process including an integrated debugging software support system. Mechanisms to achieve reliable software, which may not be directly related to testing and debugging but which support it, should be considered in addition. Requirements analysis, design, coding, verification and control techniques are all part of a comprehensive software engineering approach that should be applied during the software development phases that precede integration testing/debugging.*

One mechanism for increasing highly reliable software, directly related to debugging, is the performance requirements levied on the software through contractually specified acceptance criteria. Demonstration of acceptance criteria is achieved by formal qualification tests (FQTs). The preparation for these tests generally accounts for a large percentage of the total resources expended during the Full-Scale Development Phase. More stringent and comprehensive acceptance criteria will require extensive informal and formal testing. The debugging capabilities associated with that testing should be: (1) applicable to the software errors which are likely to occur in the particular type of software system development ** (2) provide debugging capabilities which will be usable by the debugging analyst in providing each information component in a context which is understandable to the user, (3) address problems of conserving resources so as not to provide duplicate debugging capabilities.

5.1.2 Contracting Issues

In order to present the debugging analyst with information in terms of his environment, it must be given both as *application specific information and* in the format and syntax of the abstract machine description. The procurement of software that meets these requirements may imply contracting with individual contractors for each debugging tool; with a single contractor for the entire debugging system; or the debugging system may be included with the application system and not specified as a deliverable end-item. The contract must assign the responsibility for the definition, implementation, and acceptance of the software. Maintenance activities should be considered as well. Additional decisions must be made such as where the work will be performed, what facilities are required, and what type of contract will be used. In short, the same procurement decisions must be made for the debugging system as are made for the other types of deliverable software when it is procured as a separate CPCI. For that reason and for cost considerations, it is not generally procured in that manner and many of these important issues may not receive the attention they deserve.

The use of existing, perhaps GFE, software packages is a common practice in obtaining debugging capability. However, their use may require that the contract assign responsibility for the definition, implementation, acceptance and maintenance of changes to the existing software packages to make them compatible to the environment. This requires a procurement decision for

* See Section 3.1 of Volume III for a discussion of software development methodology and techniques that can increase software reliability.

** As indicated in Section 1, this area of investigation would be a valuable adjunct to the Software Debugging Study.

assigning contractual responsibility for upgrading a previously accepted configuration item. This decision is based, in part, on whether the debugging system required is an entirely new development effort; is an upgrade to an existing hardware/software support system; must rely on a vendor supplied operating system; is using existing language compilers; etc. Determination of the existing (GFE) software components which will be made available in a given system procurement and the extent to which they are to be modified are considerations which involve analysis of the cost, performance and schedule trade-offs of the system procurement.

A decision to procure and formally accept debugging capability as part of a software support system must consider the costs required to define, produce, and maintain such a computer program configuration item (CPCI). Three important considerations should be noted:

- Testing/debugging software which must be developed to support development of embedded software will be needed in the normal maintenance activities associated with operational software. If the support software is not a contract deliverable end-item, it generally lives for only the life of the Full-Scale Development Phase, although testing/debugging activities continue through Installation, Operation, and Deployment.

- A testing/debugging software support system should be modularly designed to support ease of maintenance required for changes which will occur to augment additional functional capability as new technology warrants.

- Cost, schedule, and performance requirements for a testing/debugging support system must be defined and implemented in such a manner so as to actually be available to support the development of the application software in a timely manner.

5.1.3 Vendor-Supplied Software

Selection of specific off-the-shelf computer hardware usually implies vendor-supplied software. One of the results concluded by the site survey conducted in support of this study was that vendor-supplied debugging systems were seldom used in large-scale systems developments. This was found to be true even when some vendor-supplied software debugging capabilities appeared to be quite comprehensive. It was found that the non-use of the debugging software was attributable to one or more of the following factors:

- The debugging capabilities available are specific to the hardware environments and require the analyst to find and interpret the information in a format which may not be immediately familiar to him (i.e., the information was not given in terms of the abstract machine the analyst was using).

- The debugging capabilities were not specifically applicable to capturing and displaying the relevant information about the application software and its computing environment.

- The debugging capabilities offered by way of user's manuals may be redundant, poorly presented, and/or out-of-date with the operational software.

5.1.4 Hardware Configuration

Definition of the computer hardware configuration for system development and operations is a major procurement decision which will not be discussed in this report. While this study recognizes that hardware selection must be made in light of in-depth trade-off and feasibility studies regarding system requirements, it also recognizes that hardware with interactive processing offers capabilities which resolve many of the usability problems associated with a debugging system. If the application software's target computer does not have interactive capability, use of an interactive host development computer should be considered in relation to cost and schedule trade-offs.

5.1.5 Evaluating the Use of Development Methodology by a Contractor

One mechanism for determining the degree of relationship (i.e., integration) between the development methodology and the software engineering tools and techniques used in conjunction with it is to indirectly apply the results of studies such as this one which view one activity of the development process in relation to all other activities. This relationship can be translated to acceptance criteria to be used in source selection of proposals for Full-Scale Development contracts. Alternatively, the interrelationship of tools, techniques and development methodology can be demonstrated via the Computer Program Development Plan (CPDP). Two major considerations which indicate the degree to which software engineering is applied in an integrated manner include:

- The mechanism for demonstrating the traceability of requirements through the various system representations which evolve during the phases of the software development effort.

- The mechanisms available for controlling the technical product baseline during CPT&E. This includes definition of procedures for maintaining the history of changes to the evolving configuration item which extend beyond those required for configuration management.

A contractor's software development methodology should indicate the tools and techniques which are to be used to aid in the traceability of functional/performance requirements in the application system description. (This implies identification of the software engineering tools and techniques which directly and/or indirectly contribute to the process of translating requirements from one level of system specification to the next level of system specification.) These information components resulting from the transformation process are needed by most activities in the software development process, not just debugging. As noted in Section 3, these information

components are most effectively obtained and used in debugging when they are presented in a format and context which interrelates requirements in a syntax common to each information component.

A contractor's methods for establishing technical control over the evolving software product should contain procedures for creating, changing, and freezing the elements of the product at specific milestones in the development cycle. These procedures should be compatible with military regulations, standards and procedures, but to be truly effective, they should reflect additional technical control procedures commencing at integration-level testing. The intent of the technical control is to establish internal error/deficiency/modification procedures which restrict arbitrary changes made to an internally established technical baseline without proper authority. While these procedures are not a requirement in embedded system procurements, they are seen to be a necessary adjunct to an organized and integrated development process. In addition, the capability to define the contents of the configuration under test at any given point in integration-level testing/debugging activities is essential. This capability should include identification of the modification level of all program/data elements and a mechanism for returning the system configuration and each element within to a prior modification level at any point in time subsequent to activiation of technical control baselining.

5.2 ISSUES RELATED TO THE DESIGN AND IMPLEMENTATION OF A MODEL DEBUGGING SYSTEM

This section addresses some aspects of a design for debugging capability which make a debugging system attractive and useable, and, consequently, satisfy the requirements of debugging activities. These issues are appropriate to the development of a completely new debugging support capability or evaluation and upgrading of vendor-supplied systems or existing, project-specific software. The design aspects discussed concern obtaining applicable debugging capability by incorporating certain features in the application software design, and considerations for specific tools which might be used in the debugging system.

5.2.1 Application Software Design Features

Generally, no off-the-shelf debugging support tools can effectively display the operation of a program in terms of the abstraction created by that program. This capability requires that the application software design, the abstract machine used by the debugging analyst, and the debugging support software be integrated in such a manner so as to provide information structures relevant to the problem statement and problem solution in an understandable format. There are some design features which can be included in the design of the application software, which when implemented and used either alone or in combination with debugging tools, can more effectively provide the information requirements needed for debugging. None of these design features is particularly innovative and most are not expensive to incorporate when they are considered in the requirements and design stages of the application software. The design features which are briefly discussed below are associated with certain generic debugging tools described in Appendix A. They include:

- Programmed-In Dumps. (See Section A.4, Appendix A). This type of debugging capability is the prime means for displaying the operation of a program in terms of the abstractions created by the program. It consists of programmed instructions which output run-time information in terms created by the programmer. It is generally only used during testing/debugging because it degrades performance. It is placed in the following areas of a computer program:

  - Before and after key calculations
  - After global data is set
  - At critical decision points for control structure alternatives
  - After program entry/exit statements
  - To display key interface data
  - To provide timing and sequencing information.

- Error Messages. (See Section A.4 of Appendix A). Error messages may be viewed as a specific form of selective programmed-in dump which may, or may not, be used just for test purposes. They are used to output relevant information whenever the software encounters a probable error condition. Examples of their use include:

  - Out-of-range input values
  - Improperly formatted input values
  - Incomplete input values
  - Conflicting input requests/values
  - Operations which exceed limitations/restrictions
  - Out-of-range results calculated
  - Illegal data references
  - Run-time exceeded
  - Interface and data definition inconsistencies

- Monitor Dumps/Data Collection/Reduction. (See Section A.5 of Appendix A). This type of capability collects, records and outputs run-time information. It works most efficiently when buffers are set aside by the application software for run-time data which may be monitored in real-time by a hardware or software device. The design of this capability should consider efficiency as an important performance requirement because this type of capability can also degrade the application system performance. If the buffers are efficiently organized and consolidated to contain relevant run-time information, a large store of data may be made available for analysis.

- Auxiliary Storage/Utility Dumps. (See Section A.6 of Appendix A). Like the monitor dumps, the design of this capability should consider efficiency as an important performance requirement. It too requires the efficient organization and use of data buffers. Related data sets in the application system should be located in contiguous areas so that selection of data for display is optimized.

- Traces and Backtraces. (See Sections A.7, A.8, A.9, A.18 of Appendix A). This type of tool displays the sequence of program execution. They are most effective and useable when they provide the run-time information in terms of the control structures and associated statement labels of a program. The use of certain coding standards can enhance this capability. For example, statement labels can be used at key decision statements and interface points in a program so that the control flow may be more easily traced; if source statement line numbers are used to display trace or backtrace information, one decision statement should be used per line.

- Set-Use Matrix/Cross Reference Analysis.(See Section A.10 of Appendix A). This type of tool reflects a program's and/or a system's set/use of data and interfacing software componenets. Again, coding standards can be used to enhance its effectivity (e.g., data and programs should be meaningfully defined; data should not be indirectly referenced by a program.)

- Test Input Listers. (See Sections A.22, A.23, A.24 of Appendix A). Design consideration should be given to recording and displaying the inputs received from all devices used when the application software uses a common input processor since implementing this capability is trivial. Although test input listers may need to be deleted from the application software before delivery because they are not a requirement and/or consume resources (i.e., time and space), the capability is handy for integration testing/debugging.

- Test Beds. (See Section A.12 of Appendix A). It is often useful to be able to exercise a system in a systematic, repeatable manner. Test beds (environment simulators) can provide this capability. Essentially these tools consist of a set of hardware and software resources independent from the application system which create a dynamic environment for the application system. The input data is usually precalculated based upon simulation studies and can be re-used any number of times. The hardware used is a test bed and may be shared witn the application program (e.g., a separate job in a multiprogrammed environment) or a dedicated hardware base may be used. The following capabilities are desirable:

  - Scripted (canned) operator inputs for interactive inputs

  - Input file or data stream generation

  - Output file or data stream capture

- Programmed-In Dumps. (See Section A.4, Appendix A). This type of debugging capability is the prime means for displaying the operation of a program in terms of the abstractions created by the program. It consists of programmed instructions which output run-time information in terms created by the programmer. It is generally only used during testing/debugging because it degrades performance. It is placed in the following areas of a computer program:

  - Before and after key calculations
  - After global data is set
  - At critical decision points for control structure alternatives
  - After program entry/exit statements
  - To display key interface data
  - To provide timing and sequencing information.

- Error Messages. (See Section A.4 of Appendix A). Error messages may be viewed as a specific form of selective programmed-in dump which may, or may not, be used just for test purposes. They are used to output relevant information whenever the software encounters a probable error condition. Examples of their use include:

  - Out-of-range input values
  - Improperly formatted input values
  - Incomplete input values
  - Conflicting input requests/values
  - Operations which exceed limitations/restrictions
  - Out-of-range results calculated
  - Illegal data references
  - Run-time exceeded
  - Interface and data definition inconsistencies

- Monitor Dumps/Data Collection/Reduction. (See Section A.5 of Appendix A). This type of capability collects, records and outputs run-time information. It works most efficiently when buffers are set aside by the application software for run-time data which may be monitored in real-time by a hardware or software device. The design of this capability should consider efficiency as an important performance requirement because this type of capability can also degrade the application system performance. If the buffers are efficiently organized and consolidated to contain relevant run-time information, a large store of data may be made available for analysis.

- Auxiliary Storage/Utility Dumps. (See Section A.6 of Appendix A). Like the monitor dumps, the design of this capability should consider efficiency as an important performance requirement. It too requires the efficient organization and use of data buffers. Related data sets in the application system should be located in contiguous areas so that selection of data for display is optimized.

- Traces and Backtraces. (See Sections A.7, A.8, A.9, A.18 of Appendix A). This type of tool displays the sequence of program execution. They are most effective and useable when they provide the run-time information in terms of the control structures and associated statement labels of a program. The use of certain coding standards can enhance this capability. For example, statement labels can be used at key decision statements and interface points in a program so that the control flow may be more easily traced; if source statement line numbers are used to display trace or backtrace information, one decision statement should be used per line.

- Set-Use Matrix/Cross Reference Analysis.(See Section A.10 of Appendix A). This type of tool reflects a program's and/or a system's set/use of data and interfacing software componenets. Again, coding standards can be used to enhance its effectivity (e.g., data and programs should be meaningfully defined; data should not be indirectly referenced by a program.)

- Test Input Listers. (See Sections A.22, A.23, A.24 of Appendix A). Design consideration should be given to recording and displaying the inputs received from all devices used when the application software uses a common input processor since implementing this capability is trivial. Although test input listers may need to be deleted from the application software before delivery because they are not a requirement and/or consume resources (i.e., time and space), the capability is handy for integration testing/debugging.

- Test Beds. (See Section A.12 of Appendix A). It is often useful to be able to exercise a system in a systematic, repeatable manner. Test beds (environment simulators) can provide this capability. Essentially these tools consist of a set of hardware and software resources independent from the application system which create a dynamic environment for the application system. The input data is usually precalculated based upon simulation studies and can be re-used any number of times. The hardware used is a test bed and may be shared with the application program (e.g., a separate job in a multiprogrammed environment) or a dedicated hardware base may be used. The following capabilities are desirable:

  - Scripted (canned) operator inputs for interactive inputs

  - Input file or data stream generation

  - Output file or data stream capture

- Output data reduction

- Comparison of output data between runs.

● Emulators. (See Sections A.11, A.13 of Appendix A). In a resource bound environment, even the most primitive tools can impose unacceptable run-time overhead for debugging data collection. This problem is particularly severe for airborne computers and application dependent microprograms. One solution to this problem is the use of instruction set emulators, which interpretively execute object programs for the target computer on a general purpose host. Since real-time and space are simulated, the resources used by data collection activities can be hidden from the simulation. Such tools can allow extensive inspection of program states that are otherwise impossible. The following factors should be considered:

  - The emulator must account for physical concurrency to be effective.

  - The emulator must be a faithful copy of the original to be useful. (This can be very difficult to achieve and to verify).

  - An emulator will be very inefficient; it must reside in an environment where resources are plentiful.

5.2.2 Debugging Software Design Features

This section briefly discusses features of debugging tools which are seen to enhance the usability of these tools.

● Debugging Command Language. The syntax used in interactive systems for specifying support software capabilities, especially for debugging, should be comprehensive, consistent, and easy to use and remember. Commands are necessary for requesting displays, traces, stop/continue program execution, set data, specify conditions, remove commands, specify locations/names, dump locations, etc.

● Flexibility in Specifying Output Devices. For both batch and interactive systems, the debugging tool's nominal output devices and their associated attributes should allow for specifying variations in length of storage, timing parameters, and alternative output devices, etc.

● Flexibility Options in Recording/Reduction Tools. (See Section A.23 of Appendix A). Tools which record run-time information during program execution and later output it for analysis should allow for variations in the format to be used during the reduction of the data. For example, character data might be output in a format different from numerical data. Allowing user specification of a subset of captured data is also useful.

- System Status Information. A mechanism is needed which provides identification of all elements in the hardware/software configuration used during each program test execution. The information should include both the name, version and/or modification number of all software components in the application software and abstract machine. Obtaining current information regarding the application software configuration requires that the design for the software which captures, records, and displays the information be correlated with the development methodology which defines procedures entering, changing and deleting elements in the data base.

- Representation of Data. The run-time information collected and displayed by debugging tools should be in terms of the application system description and abstract machines used. This requires interfacing the abstract machine used by the programmer (i.e., the HOL) with the abstract machine used by the debugging analyst.

- Reliability/Maintainability of Debugging Software. The design for the debugging software should be as carefully engineered as the design of the application software. It is recommended that the debugging software be formally accepted in order to ensure that its operational performance meets reliability and maintainability requirements. These type of requirements can be achieved only when specific features (e.g., top-down) are incorporated within the design and implementation approaches.

6. RECOMMENDATIONS FOR RELATED STUDIES

A software debugging methodology must necessarily be integrated with the methodologies applied to other software development activities which both precede and succeed debugging. This is due to the fact that debugging is the process of isolating and resolving errors introduced into the configuration item by any one of the other development activities. In addition, many of the information components required by debugging are produced in previous development activities, and information resulting from debugging activities may change one or more of those information components. For that reason, debugging cannot be viewed in isolation, and the results of this study should be incorporated with results of other study efforts in order to provide an integrated software engineering approach to software development. The recommendations for additional work are based on this premise and are briefly described below. Most of these recommendations have been referred to at appropriate places in the preceding text. They were not elaborated upon in this study since they were beyond the intended scope of integration-level debugging methodology.

- A Study and Correlation of Error Symptoms/Error Causes. The purpose of this study is to conduct an investigation of generic error symptoms associated with specific application software which manifested themselves during integration testing of hierarchically developed computer program configuration items. The error symptoms will be initially described via a software discrepancy report (SDR), and tracked during the debugging process and correction implementation process. The results of these two processes should be analyzed in an attempt to produce: (1) a correlation of error cause with error symptom; (2) an indication of the development activity and specification (requirements, design, source program) in which the error originated; and (3) a correlation of error cause and application. If a correlation of error symptoms with error causes could be established within generic categories of applications for embedded software, a more efficient debugging process would result. In addition, early detection of requirements and design errors will result in significant savings of resources, since repair of these types of errors increase in cost as the development cycle progresses.

- Use of Assertions with Program Design Notations. The use of assertions in the debugging methodology is an informal variant of the use of precisely specified assertions in formal proof of program correctness. An informal assertion is a statement of the assumptions that a program component makes about its environment. This concept can be readily applied in the design process, especially if a program design notation is used. These notations allow specification of algorithms in terms of abstract (incompletely specified) components and the operations defined on them. The designer's assumptions about an abstract component's environment can be formulated as assertions, as can the designer's view of the effect of operations upon an abstract component. If such assertions were provided in a textually distinguished manner, they could easily be extracted and used for guidance in inter-component testing and debugging.

Normally, an entry assertion is provided for each operation defined on a component which states the constraints on the state of the component which must hold in order for the operation to perform correctly. An exit assertion describes the effect of the operation upon the state of the component. Once the component is fully specified (in terms of lower level components), it must be tested and debugged. The following situation must be verified:

entry assertion ⟶ operation ⟶ exit assertion

meaning that if the entry assertion holds before performance of the operation, the exit assertion will hold after performance of the operation. This relation can be tested without knowledge of the detailed specification of the operation. However, if the relation does not hold, the operation must be debugged. Debugging a faulty operation requires full access to the specification of the data structure it operates upon, and the specification of the operation itself. The specification must be examined to determine how the operation fails to support the entry ⟶ exit relation. Further, if a system is being debugged in which the entry and exit assertions are not supplied with the system's description, it is often necessary to inspect the component's specification in order to determine what the assertions are. The use of entry and exit assertions in the debugging process model was described in Section 3.3.3.

If the overall system is nearly decomposable*, the correctness of a component can be determined by an examination of the above described nature. If the component is itself part of a larger component, it can be viewed as a primitive operation of the larger component which is used as part of the detailed specification of the larger component. This next larger component has its own entry and exit assertions about its environment, and its own specifications of operations and data in terms of the smaller components it uses. The implication of this structure is that nearly decomposable systems can be tested and debugged one component at a time. Note that this statement relies upon a hierarchic view of components, i.e., a high level component may "own" many smaller components.

- Use of Abstract Data Types with Program Design Notation. The use of subroutines in decomposing a complex program into a hierarchy of abstract operations has become widely accepted. This technique is useful within a wide range of source languages, and is easily suggested in a program design notation. Similar structuring concepts are just beginning to gain acceptance when applied to decomposing a program's data space into a hierarchy of abstract data "types". The class construct of Simula 67 has been used as a model for the use of abstract data types in programming languages (as in Concurrent

* A nearly decomposable system is one in which a component's function can be understood largely from local context, with weak, but well defined, interaction with other components.

Pascal, and CLU). Unfortunately, this concept is not readily implemented in currently popular implementation languages (such as FORTRAN and JOVIAL), or even in currently advocated program design notations. The combination of a hierarchy of abstract data types and a hierarchy of abstract operations defined on them is a powerful mechanism for achieving nearly decomposable system structure. The implication for program verification and debugging is that the components of such a system can be examined one at a time, with the assurance that components only interact in well-defined ways. The suggested research in this area would address the inclusion of abstract data types in a program design notation; and the identification of a well defined, step by step, methodology for transforming a design expressed in this manner into a source program implemented in current generation programming languages.

- Integrated Methodologies for Tool/Technique Utilization Within the RADC Environment. As the results of the debugging methodology indicate, there is a need to integrate and coordinate the use of tools/techniques with each of the development activities. There exists at RADC tools which can be used for each of the major development activities: requirements, design, code, test, debug, management. Yet there is no generic methodology which suggests the use of such tools in an organized and controlled basis. For example, tools such as URL/URA, Design Language Notation, Automatic Verification Systems (FAVS/JAVS), and the program support library (PSL) could be extended and integrated in such a manner as to provide traceability from requirements through acceptance testing for a CPCI. Further, ad hoc tools usage does not provide the software development process with the control and visibility it requires to produce quality systems. A methodology which defines procedures for manual and automated techniques for control and visibility will result in a higher quality software product. The development of an integrated methodology can also suggest the acquisition of new tools that can augment the existing RADC tools.

- Feasibility Analysis/Technical Development Plan Utilizing the NSW as a Software Engineering Facility. The long-term goals of RADC must consider extending the concept of the NSW to provide its user community with innovative software engineering techniques in order to test their validity, viability, and applicability to embedded software development. The ever-increasing costs of development and maintenance of software require that quality characteristics for computer program configuration be specified, designed, implemented, and maintained in each of the respective life cycle phases. The development of integrated methodologies for tool/technique utilization within the RADC environment, described above, is the first step in establishing a software engineering facility that would provide such tool/techniques. The next step would be to determine the feasibility of applying facilities of the NSW to support the integrated methodologies: that is, to determine whether the NSW can support the use of URL/URA, Design Language Notation, program support library (PSL), etc., and provide the underlying framework

for their integrated use. If the application of the NSW appears feasible, then the third step would be the generation of a technical development plan for establishing NSW as a software engineering facility which supports the integrated methodologies. The technical development plan will identify the products that need to be added to the NSW, the tasks required to develop the products and the resources and schedule required to achieve them.

APPENDIX A: Generic Debugging Tool Requirements

This Appendix describes generic types of debugging tools that can constitute a debugging system. The specific requirements for each tool to support its use in generating, analyzing or maintaining the information needed by the debugging process model are discussed under the following subsections:

- Abstract Machine Representation. The requirements for a debugging tool to provide program state information in a representation appropriate to a specific level of abstract machine are given. The abstract machines considered include:

  - Micro-Programmable Computer
  - Basic Computer
  - Operating System
  - Assembler
  - Macro Library
  - Compiler/Interpreter
  - Subroutines

- Capability and Usability Features. The specific features of a debugging tool which makes it comprehensive and easy to use and access are given.

- Program State Information. The requirements for a debugging tool to provide one or more of the several components of program state information are given. The components are described in Section 2.3.1 and include:

  - Symptom Locality
  - Traversed Paths
  - Name-Space Values
  - External System Status
  - Simultaneous Events
  - Space Allocation and Timing

Other important features, capabilities and requirements of these tools not directly related to the debugging process model are discussed in greater detail in Section 3.3.2 of Volume III ("Literature and Site Surveys - Interim Report of the Software Debugging Methodology Study - Final"). The Volume III descriptions introduce the tools by discussing their general features, as well as describing how other debugging considerations (software tolerance requirements, language considerations, interactive versus batch systems, system hardware/software architecture, and software errors) relate to the tools.

A.1 POST-MORTEM DUMPS

Post-mortem dumps provide the static state of the instructions and data of the abstract machine at a particular point in time when a program is not executing.

A.1.1 Abstract Machine Representation

This tool should provide program state information in the representation described for each of the following abstract machines:

- Basic Computer. Program state information should consist of machine register contents, current location counter,memory addresses and contents. The address of the location causing an illegal termination should be given.

- Operating System. For post-mortem dumps of programs run under an operating system, displays of schedule queues, resident programs, memory maps, corrector lists, and compool lists should be provided.

- Assembler. Post-mortem dumps of programs written in an assembly language should provide memory locations in terms of statement and data labels, or relative addresses. Memory contents should be provided in terms defined by assembly language data declarations (e.g., integer, floating point, hexadecimal).

- Macro Library. For post-mortem dumps of programs using macro instructions, the program instructions which are generated by macro instructions should be displayed together with the macro instruction and its input/output parameters.

- Compiler/Interpreter. For post-mortem dumps of programs coded in a higher order language (HOL), the output should be in terms of the HOL processing instructions.

- Subroutines. For post-mortem dumps of programs using subroutines, the program instructions which are related to the subroutine should be displayed together with the subroutine name and its input/output parameters.

A.1.2 Capability and Usability Features

This tool should provide the following features:

- Time-of-Output. The capability to obtain a memory dump before and after normal termination of a program, and at time of abnormal termination should be available.

- Selectivity of Output. A means of specifying dump of all main memory, of selective programs or data, and of ranges of locations should be available. The specification language should be in terms of the abstract machine representation.

- Batch Systems. All output should be to devices whose output is available to the programmer.

- Interactive Systems. Large memory dumps should be to devices accommodating large output, i.e., the printer. Selective small output should be available on interactive devices, e.g., CRT terminal. Hardcopy output of CRT information should be a capability.

- Real-Time Systems. The capability for post-mortem dumps before and after normal termination of a program is not feasible, but abnormal termination dumps should exist.

A.1.3. Program State Information

This tool should provide the following components of program state information in the abstract machine representations given in A.1.1.

- Symptom Locality. The area of abnormal termination (if one occurred) should accompany a post-mortem dump.

- Name-Space Values. The static state of all program name-space values which occupy primary memory at the time of the dump should be provided.

- External System Status. The interface information of all external hardware and software which is available at the time of the dump should be provided.

- Space Allocation and Timing. Post-mortem dumps should show how space is allocated in primary memory at the time of the dump. No timing information is provided.

A.2 SNAPSHOP DUMPS

Snapshot dumps provide the dynamic state of data values of the abstract machine while a program is executing.

A.2.1 Abstract Machine Representation

This tool should provide program state information in the representation described for each of the following abstract machines:

- Basic Computer. Program state information should consist of memory addresses and contents.

- Operating System. Snapshot dumps of disc and other I/O data transfer buffers should be provided.

- Assembler. Snapshot dumps of data should provide memory location in terms of statement and data labels or relative address. Memory contents should be given in terms defined by assembly language data declaration.

- Compiler/Interpreter. Snapshop dumps of data should be in terms of HOL data labels and HOL-defined data formats.

A.2.2 Capability and Usability Features

This tool should provide the following features:

- Dump Data Specification. There should be a means of specifying individual values (memory locations, data variables, parameters) or areas (memory location ranges, tables, arrays, data blocks, buffers) to be dumped.

- Limiting Output. Output from snapshop dumps can be large in quantity in some cases; for example, if item or area to be dumped is repeatedly output during the execution of a program, a means should be provided to limit output when such problems can be envisioned. The capability to request that only changes be output, to specify the frequency of output or to specify a start and stopping point for output, should be considered.

- Multiple/Overlapping Requests. The capability should exist to request display of more than one non-contiguous value or area at a time. Cases where areas overlap should result in a single display of the merged requests.

- Input/Output Devices. CRT terminals are ideally suited for request and display of snapshopts in an interactive environment. The typewriter is suitable for requests but not for display because of the slowness of its operation. The card reader/printer can be used for request and display in a batch environment.

- Time Tagging. The output of a snapshot does not occur simultaneously with its request, but follows it at some point in time. Also, the data shown could change its value in time. For this reason, snapshots should be time-tagged.

- Dynamic Snapshots. Changing data should be highlighted (e.g., brighter CRT characters) to distinguish it from data in the snapshot which remains unchanged.

A.2.3 Program State Information Requirements.

This tool should provide the following components of program state information in the abstract machine representations given:

- Name-Space Values. The dynamic state of all program name-space values which occupy primary memory at the time of the snapshot should be provided.

- External System Status. The interface information of all external hardware and software which is available at the time of the snapshot should be provided.

- Simultaneous Events. The use of simultaneous snapshot dumps of simultaneous processing is practical only when hardcopy output devices are used to note the simultaneity. Also the time required for the dump processing and output can affect the actual simultaneous recording of data.

A.3 BREAKPOINT TRAP DUMPS

Breakpoint/Trap dumps provide a dump of the abstract machine at a particular point in a program's execution. The breakpoint/trap dump occurs when a certain location or condition is met, and requested data is displayed. The execution may or may not continue after the dump. Breakpoint/trap dumps represent a consolidation of three debugging tools addressed separately in this appendix, i.e., (1) software and hardware breakpoint/traps (section A.16 and A.17 below) establish and detect the certain condition, desired before the dump is made, (2) post-mortem dumps (section A.1 above) perform a dump when the condition exists and the program execution is halted, and (3) snapshot dumps (section A.2 above) perform a dump when the condition exists but program execution is not to be halted. The combined factors indicated for these tools apply to breakpoint/trap dumps.

A.4 PROGRAMMED-IN DUMPS

Programmed-in dumps provide the dynamic state of the abstract machine in terms of the data abstractions defined in the program.

A.4.1 Abstract Machine Representation

This tool should provide program state information in the representation described for each of the following abstract machines:

- Operating System. Operating systems should provide the following via programmed-in code: disc data handler diagnostics, I/O handler diagnostics, interrupt diagnostics, program execution timing data, memory maps, and system status.

- Assembler. Programmed-in dumps from programs written in assembly language should be in the format defined by the programmer.

- Compiler/interpreter. Programmed-in dumps should be in terms of names and values defined by the programmer. The representation could be in terms of HOL variable names with informative comments.

A.4.2 Capability and Usability Features.

This tool should provide the following features:

- Compiler Capabilities. Existence of a DEBUG capability in the compiler similar to FORTRAN DEBUG, makes the generation and activation of programmed-in code a compiler/operating system function. This eases the coding of programmed-in dumps, but in some cases the capability is too restricted and, therefore, tends not to be used. This type of capability should be carefully considered because it results in debug code being generated at coding time instead of at debug time. However, its usability aspects should also be considered and alternative means provided if the compiler's capability is too restricted or difficult to use.

- Common Debug Routine. The use of a common routine for the control of programmed-in dumps is another option. Such a routine determines whether a switch has been activated and, if so, PRINT statements can be executed. The PRINT statement can be in the common routine or it can be outside. The latter case enables one request to be made for output for multiple PRINT statements.

- Tool Activation. Regardless of whether the programmed-in dump capability is a function of the compiler, a common routine, or simply an individual PRINT statements, a means should exist for activation/deactivation of the tool.

- Tool Deletion. One of the advantages of the programmed-in dump capability being provided via the compiler is the capability to have the compiler recompile the source code and ignore the DEBUG statements. This provides a reliable way of deleting debug code when space and timing requirements require it to not exist in operational programs.

- Limiting Output. When programmed-in dumps are used to print information at a location which is frequently operated (e.g., within a computational loop), the output may be unnecessarily excessive. To prevent this, the source code itself should be used to limit the output by creating conditions to be met before anything is printed or to select the frequency of output. In some cases, such conditions can be built into common routine used with the tool.

- Recompilation. For cases where the debugging analyst finds it necessary to insert debug code into a program, it is important that recompilation is made as easy as possible. For interactive system, a partial recompilation capability is most efficient.

A.4.3. Program State Information Requirements

This tool should provide the following components of program state information in the abstract machine representations given:

- Symptom Locality. Considering error messages as a special form of programmed-in dump, this tool can provide information on the locality of occurrence of the following primary symptoms: error messages themselves, abnormal initiation, abnormal termination, inputs not accepted, incorrect format/sequence, excessive output, and excessive run time.

- Traversed Paths. Messages can be printed when certain areas of code are entered.

- Name-Space Value Settings. Any program name-space value which occupies primary memory can be provided during a program's execution.

- External System Status. The interface and status information of any external hardware or software which is available during a program's execution can be provided.

- Simultaneous Events. Simultaneous programmed-in dumps can be obtained to determine simultaneous events, but the effects of the timing perturbations of the tool itself must be considered.

- Space Allocation and Timing. A system routine can be called by programmed-in code to provide output of a memory map indicating storage allocation.

  Likewise, system timing routines may be called to provide program execution timing.

A.5 MONITOR DUMPS

Monitor dumps provide the dynamic state of a subset of the abstract machine levels, i.e., the micro-programmable computer and the basic computer.

A.5.1 Abstract Machine Representation

This tool should provide program state information in the representation described for each of the following abstract machines:

- Micro-Programmable Computer. Monitor dumps at this level should be in terms of the values of flip-flops, data bus, registers, micro-instruction addresses, end conditions, and symbolic functions.

- Basic Computer. Monitor dumps at this level should be in terms of machine register contents, location counters, memory addresses, and memory contents.

A.5.2 Capability and Usability Factors

This tool should provide the following features:

- Output Sources. Output generated directly from hardware is usually difficult to read and analyze. The output should be reformated and displayed on typical computer peripheral devices.

- Dump Area Specification. If the monitor device is strictly a hardware tool, absolute locations of start and stop of dump should be the means provided to specify the dump area. If inputs can be software controlled, the use of source language label specification should be available.

- Output Quantity. The amount of the output produced by monitor dumps should be restricted by a combination of readability and elapsed time considerations. The data display should be readable and current.

- Dynamic Data. Monitor dumps are essentially the same as snapshot dumps in this regard. Features such as time-tagging and data high lighting, should be considered.

A.5.3 Program State Information Requirements.

This tool should provide the following components of program state information in the abstract machine representations given:

- Name-Space Value Settings. Any program name space values which occupies primary memory during a program's execution should be provided.

- External System Status. Information on the state and status of all external hardware and software should be provided.

- Simultaneous Events. The simultaneous setting of a program's name-space values, or of that of two or more programs, should be provided.

A.6 AUXILIARY STORAGE/UTILITY DUMPS

Auxiliary storage utility dumps, which will be referred to as secondary dumps, provide the status of the secondary storage and selected data values at a point in time when a program is not executing.

A.6.1 Abstract Machine Representation

This tool should provide program state information in the representation described for each of the following abstract machines:

- Basic Computer. Considering secondary storage as an extension of the basic computer, secondary dumps at this level should be in terms of the hardware addressing of the storage devices. For example, disc dumps should be in such terms as blocks, cylinders, tracks, and sectors.

- Operating System. Data for secondary storage created by the operating system should be provided in the terms used by the operating system software.

- Assembler. Dumps of secondary storage data created by an assembly language program should be in terms of the assembly data labels or relative addresses. The format of values displayed should be the same as those created by the assemble language program. Dumps of assembly language programs stored on disk should be in terms of the original source code, i.e., statement labels, symbolic operations, modifiers and operands.

- Macro-Library. A segment of the macro-library usually supports the application programs with the means of input and output of secondary storage data. Dumps of such data should be in the same terms as that of the program, with which the macro-library is used, is written (i.e., assembler and compiler/interpreter).

- Compiler/Interpreter. Secondary dumps of data created at this level should be in terms of the HOL data definitions of the data, i.e., HOL defined data value units. Dumps of HOL programs stored on secondary storage should be in terms of the original source language, i.e., HOL statement labels, data definitions and processing operations.

- Subroutines. Similar to micro library.

A.6.2 *Capability and Usability Features*.

This tool should provide the following features:

- Display Sources. The secondary dump capability should be provided for all secondary storage data in the computer system. This includes data stored on disc, drum, tape, and paper tape.

- Dump Requests. Dumps of secondary storage should provide for the selection of data components (data specific locations, and assembly, HOL, or Compool data items, variables, constants, tables, arrays, blocks, record, files, areas, etc.) or the entire storage capacity. The specification language should be in terms of the abstract machine representation.

- User Defined Display Formatters. A capability should be provided for the users to define their own output formats for secondary disc dumps. This includes defining the nominal output devices, page format, including headings, labels and data value units, as well as defining units conversion algorithms. Formats should be capable of being modified, added or deleted at any time by the user.

- Logs. Logs are a summary of the data on a secondary storage device. Actual data values are not displayed. For example, a tape log might show the tape header records, and then list the file and records of the tape and simply provide the length of these elements. The log capability is a very useful feature for isolating data for which more detail is needed.

A.6.3 Program State Information Requirements

This tool should provide the following components of program state information in the abstract machine representations given:

- Name-Space Value Settings. The state of a program's name-space values which occupy secondary memory before or after the program's execution should be provided.

- External System Status. Secondary dumps provide a means to display the inputs from external systems as well as the outputs to those systems. Software interfaces with external systems in the form of tape, disc, card, or paper tape storage are made visible.

- Space Allocation and Timing. The allocation of secondary storage should be provided.

A.7 DYNAMIC INTERNAL TRACES

Dynamic Internal traces operate within the abstract machine and display the traversed paths of the abstract machine as a program is executing.

A.7.1 Abstract Machine Representation.

This tool should provide program state information in the representation described for each of the following abstract machines:

- Basic Computer. The trace should be provided in terms of absolute address executed. For a full instruction trace, the program state representation should also consist of the following for each selected location traced: machine register contents, memory address of instruction being obeyed and selected memory contents.

- Operating System. The trace should be in terms of the operating system elements operated, and possibly their input and output values. The trace should also include the name of each operating system routine called before and after the operation of the application program.

- Assembler. The trace should be in terms of assembly code instruction statement labels executed or the assembly language instruction executed, and possibly the values of all data operands and newly assigned values.

- Macro Library. The trace should be in terms of the macros operated, and possibly their actual input and output values.

- Compiler/Interpreter. The trace should be in terms of HOL source statement numbers executed, HOL instruction statement labels executed, or the HOL statements themselves, and possibly the values of all variables and function evaluations required for expression evaluation. In some cases, the trace may also display any newly assigned values, the outcomes of conditional tests, procedure calls, and the correspondence between formal and actual parameters.

- Subroutines. The trace should be in terms of the subroutines operated, and possibly their actual input and output values.

A.7.2 Capability and Usability Features.

This tool should provide the following trace options:

- Location Trace. This type of trace provides the location or source statement numbers of all instructions executed during a program's operations. The advantage over a jump trace is a complete picture of the traversed path is presented. The disadvantages are excessive output and increased operating time.

- Jump Traces. This type of trace provides the location or source statement numbers of all instructions which result in a jump to a non-sequential location during a program's operation. The advantage is that an indication of the traversed paths of a program is presented with the minimum display data necessary. The disadvantage is that program state information over the entire traversed paths is not provided.

- Reference Trace. This type of trace is the same as the trap tool described in Sections A.16 through A.17. Each time a particular location, instruction, or statement, or a particular condition is encountered, or particular data is referenced, a display of selected data occurs.

- Macro Library and Subroutine Traces. Traces of machines can be provided by the use of programmed-in code (which can be switched on/off) within macros or subroutines which indicate execution has started/stopped and the input/output parameters. (See A.4, Programmed-In Dumps). In addition, the following features should be provided:

- Trace Mode Interrupts. The trace mode interrupt capability of most software systems has no selectivity features associated with it. Thus, even if the instruction being executed is not within some location bounds specified by the user, if the trace mode is "on", an interrupt will occur. This causes trace to be time consuming even when the limits of the trace are minimal. The design of trace tools and operating systems should provide schemes to minimize this effect as much as possible.

- Limiting Output. The execution time of trace tools can be minimized by providing the user the means of limiting the amount of output via the following methods:

  a. Abbreviation. Abbreviating the data output.

  b. Repeat Counting. Checking to see if the next group of trace information to be output is the same as the previous, and bypassing its display if this is true. An example of this is the case where a DO or FOR loop is executing the same group of instructions until a condition is met. This repeating of this information usually is not important to the debugging analyst.

  c. Bounding. Providing the capability to specify one or more areas to be traced, instead of the entire program's operation.

  d. Output Item Selection. Allowing user to specify what value or types of values should be displayed.

A.7.3 Program State Information Requirements

This tool should provide the following components of program state information in the abstract machine representations given:

- Traversed Paths. This tool provides all the information necessary to determine the traversed paths of a program.

- Name-Space Value Settings. This tool should provide the dynamic state of selected program name-space values connected with the execution of each instruction or source statement of a program that was traced.

- External System Status. This tool should provide the interface information of all external hardware and software which is available at the time a traced instruction or source code statement is executed.

A.8 RECORDED TRACES

Recorded traces operate within the abstract machine and record the traversed paths of the abstract machine as a program is executing. The recorded traversed path information is stored and can be processed and displayed as indicated in Section A.8.2 after the program has finished executing.

A.8.1 Abstract Machine Representation

The abstraction machine representation for recorded traces is the same as for Dynamic internal traces (Section A.7.1).

A.8.2 Capability and Usability Features

This tool should provide the following features:

- Dynamic Internal Trace Features. Recorded traces should possess all of the capability and usability features described for Dynamic internal trace in Section A.7.2.

- Execution Speed Control. This tool should provide the user with the capability of controlling a program's apparent execution time, by slowing-down and speeding-up of operation of the trace.

- Reinitiation. This tool should provide the capability for the user to discontinue the current trace request and reinitiate the trace beginning at selective locations.

- Recording/Display Selectivity. The user should be able to select the data displayed from the total amount of recorded data as each instruction executes. Specification should be in terms of the abstract machine, and should include start and stop spans for both recording and display.

A.8.3 Program State Information Requirements.

The program state information requirements for recorded traces are the same as for dynamic internal traces (Section A.7.3).

A.9 MONITOR TRACES

Monitor traces operate external to the abstract machine and display, or record and later display, the traversed paths of the abstract machine as a program is executing. They differ from dynamic internal and recorded traces in that they don't perturb the space environment of the abstract machine.

A.9.1 Abstract Machine Representation

The abstract machine representation for monitor traces is the same as for dynamic internal and recorded traces (Section A.7.1).

A.9.2 Capability and Usability Features

The capabilities and usability features associated with monitor traces are the same as both the dynamic internal and the recorded trace features (Sections A.7.2 and A.8.2).

A.9.3 Program State Information Requirements

The program state information requirements for monitor traces are the same as for dynamic internal and recorded traces (Section A.7.3).

A.10 SET-USE MATRIX/CROSS REFERENCE ANALYSIS TOOL

The Set-Use Matrix/Cross Reference Analysis tool provides a static view of the relationships of the program state representations within and between various abstract machines.

A.10.1 Abstract Machine Representation

For abstract machines for which this tool is applicable, the tool should indicate: (1) all elements of the abstract machine set/used within the same abstract machine, and the setting and using elements, (2) all elements of other abstract machines set/used by this abstract machine, and (3) all elements of the other abstract machines which set/use elements of this abstract machine, and the setting and using elements. The elements indicated should include all routines, source statement labels, data items, constants, tables, arrays, or blocks referenced by or within an abstract machine. The following summarizes the abstract machines applicable to thic tool (Section A.10.2 presents details for each particular type of this tool).

- <u>Operating System</u>.
- <u>Assembler</u>
- <u>Macro Library</u>.
- <u>Compiler/Interpreter</u>.
- <u>Subroutines</u>.

A.10.2 Capability and Usability Features

This tool should provide the following features:

- Program Cross Reference. A cross reference should be provided for each application program in the system. This information should be provided by the system's compiler or assembler or by a stand-alone program operating on the application program's source code.

  - Assembly Language Cross Reference. An alphabetically sorted list of all labels and names within a program should be provided. For each entry in the list, the following information should be given.

    a. Source sequence number of the instruction in which the label or name is declared or defined, as appropriate.

    b. Relative address of the label or name in respect to some standard location. Also, an indication of whether the address is in the system data area or the procedure area or whether the address is absolute.

c. The source sequence number of all instructions in which the label or name is referenced, and how referenced (set, used, both).

- HOL Cross Reference. An alphabetically sorted list of all labels and names referenced with a HOL program should be provided. For each entry in the list, the following information should be given.

    a. An indication of where the label or name is defined, e.g., the system compool, the main program area, within a procedure, etc.

    b. An indication of the class the label or name belongs to, e.g., it is an item, procedures, data block, array, table, table item, etc.

    c. The HOL source statement number where the label or name is declared or defined, if appropriate.

    d. The way the label or name was used (entered, used, set, or not referenced) followed by the source statement numbers where this action occurred.

- Computer System Cross Reference. A cross reference should be provided for all system level programs and common data (including compool defined data) which are eligible for reference by other system level programs or data. An individual, alphabetically sorted list should be provided for each of the following system elements.

    - System programs, subroutines, or macros

    - System data blocks.

    - System items, variables, or named-constants

    - System tables or arrays

  For each entry in each such list, an indication of what system programs, subroutines, or macros reference the entry should be given, together with the type of references (entered, set, used, both, or not referenced).

A.10.3 Program State Information Requirements

This tool should provide the following components of program state information in the abstract machine representations given:

- Symptom Locality. This tool supplies candidate localities for the occurrence of an error message by indicating which statement references the data structure containing the message, or by indicating where a system error message routine is used. It can also supply candidate locations where data is set or output incorrectly and where inputs are not accepted or are processed incorrectly.

- Name-Space Values. This tool provides a static view of the setting and use of all name-space values within an abstract machine. The representation is not necessarily that of the actual settings and uses which occur when the software is executed, however.

A.11 HARDWARE MONITORS

Hardware monitors operate external to the abstract machines and provide, or aid in providing, dynamic program state information for a subset of the abstract machine levels, i.e., the micro-programmable and basic computer levels.

A.11.1 Abstract Machine Representation

This tool should provide program state information in the representation described for each of the following abstract machines:

- Micro-programmable computer. This tool should provide program state information in terms of registers, data and address lines, as well as control, sequencing, and interrupt signals. Request specification should be in terms of absolute instruction, data, or address values.

- Basic Computer. This tool should provide program state information in terms of machine register contents, location counters, memory addresses, and memory contents. Request specification should be in terms of absolute instruction, data, or address values.

A.11.2 Capability and Usability Features

This tool should provide the following features:

- Instruction Trapping. Synchronizing on a specific instruction and sequentially displaying subsequent instructions.

- Interrupt Triggering. Triggering an interrupt after execution of each instruction in the user program. (Interrupt service software can then recover the address and print it with the associated machine's registers, before returning to program execution.)

- Location Monitoring. Providing monitoring of specific locations or data words by storing bus information of each occurrence.

- History Collections. Storing a continuous history of information and halting on a specific condition.

- Code Timing. Timing selected sequences of program code.

- Counting. Event counting (e.g., accesses to an address or peripheral.).

- Address Referencing. Collecting sets of addresses or instructions which refer to a specific location.

- Bus Storage. Specifying a set of addresses which will trigger bus data storage.

- Hardware Simulation. Simulating non-existent hardware.

- Clock Simulation. Replacing the system clock.

- Microcode. Employing microprogramming for program monitoring.

- Rate Control. Slowing down and speeding up the execution rate of the computer.

- Execution Control. Initiating, halting, continuing, and altering sequence of execution.

- Altering Memory Contents. Setting memory with data or instructions.

- Instruction Stepping. A means to execute instructions one-at-a-time.

A.11.3 Program State Information Requirements

This tool should provide the following components of program state information in the representation given in A.11.1.

- Name-Space Values. The dynamic state of all name-space values which occupy primary memory during a program's execution are provided.

- External System Status. Information on the status or states of all external hardware and software can be provided.

- Simultaneous Events. The simultaneous setting or accessing of a program's name-space values, or of that of two or more programs can be provided.

- Space Allocation and Timing. Timing selected sequences of program code can be provided.

A.12 MONITOR COMPUTERS

Monitor computers operate outside the abstract machines and provide for monitoring the program state information of the abstract machines and of simulating the input and monitoring the outputs of the abstract machines.

A.12.1 Abstract Machine Representation

This tool should provide program state information in the representation described for each of the following abstract machines:

- Basic Computer. Program state information can be provided in terms of memory addresses and memory contents.

- Assembler. Data variables, constants, and entry points should be provided in terms of the definitions, labels and names used in the assembly language program.

- Macro Library. Data variables, constants, and entry points should be provided in terms of the definitions, labels and names used in the HOL code program.

A.12.2 Capability and Usability Features

This tool should provide the following features:

- Interactive Commands. The monitor computer should be an interactive system and provide the user with display, tracing, modification and program control capabilities via commands such as the following:

COMMAND	ACTION
DUMP	Display all or portions of a program's data or code.
TRACE	Output information on sequence of program's execution of code.
RETRACE	Trace backwards from current statement being executed.
GO	Operate the program.
STOP	Cease operation of the program.
SLOW	Slow down execution speed of the program.
FAST	Speed up execution speed of the program.
SET	Set of modify program's data or code.
IF	Perform debugging action if certain conditions are met.
AT	Perform debugging action at certain statement or location.
REMOVE	Discontinue current debugging action.

- Non-interactive Computer-to-Computer Communication. The monitor computer may simply observe and not perturb the contents of the host computer's memory. This is accomplished via a hardware interface unit which attaches to the host's memory interface channel.

- Interactive Computer-to-Computer Communication. For commands requiring control of the host computer's program execution, it is usually necessary to use the interrupt system between the computers. Control is maintained and exercised via the use of interrupts in both directions.

- Higher Level Language Symbols. In order to provide for abstract machine representation for levels at or above the assembler level, the compiler/assembler usually must generate a symbol table containing all variable names and entry point names in a program. The monitor computer must have this symbol table available for display and input request processing.

- Input/Output Processing. The monitor computer can be programmed to simulate real time events (e.g., peripheral transfers and interrupts) and perform analysis and evaluation of the host computer's outputs. This open and closed loop capability allows external system status information to be provided for debugging purposes.

- Inter-Computer Breakpointing. Conditional halts can be employed by the monitor computer to stop execution of the host machine and return to the user whenever a specific set of conditions holds. The advantage of this feature lies in the fact that the conditions are evaluated during each instruction cycle at run time, as opposed to the more traditional breakpoint insertion for real-time software which is effectively performed at load time.

- Time and Frequency Information. Important information may be retrieved by halting the computer under test, retrieving the program counter and instruction register, saving these in a trace buffer each time the program counter is changed, and allowing the computer under test to execute the instruction. At the end of a test run, the buffer is processed by a program that generates execution time and frequency information.

A.12.3 Program State Information Requirements

This tool should provide the following components of program state information in the abstract machine representations given in A.12.1.

- Traversed Paths. This information can be provided in terms of the abstract machine representation.

- Name-Space Values. The dynamic state of all program name-space values which occupy the host's primary memory can be provided.

- External System Status. Information on the status and status of simulated external hardware and software can be provided.

- Space Allocation and Timing. Timing of instruction execution can be provided.

A.13 COMPUTER EMULATORS

A computer emulator provides a lower level of abstract machine which can be used to monitor the next higher levels of abstract machines.

A.13.1 Abstract Machine Representations

This tool should provide program state information in the representation described for each of the following abstract machines:

- Micro-programmable computer. Program state information should be provided in terms of the microcode of the control store.

- Basic Computer. Program state information should be provided in terms of machine register contents, location counters, memory addresses, and memory contents.

A.13.2 Capability and Usability Features

This tool should provide the following features:

- Microcoded Debugging. The control memory of an emulator can be programmed to produce, record, or display debugging information on the user instructions (program memory instruction) or the microinstruction themselves (control memory). This is particularly applicable to an emulator whose control memory is writeable (writeable control store), as opposed to readable-only (read-only memory-ROM).

- Environment Simulator Coupling. An emulator can be coupled to an environment simulator (e.g., a monitor computer via a hardware interface device) to furnish it with the required inputs to process and to display the emulator's outputs. This emulator/environment simulator approach is especially useful for debugging avionics flight computer software, because these flight computers provide outputs only to interfacing avionics hardware, not to the standard I/O peripherals.

A.13.3 Program State Information Requirements

This tool should provide the following components of program state information in the abstract machine representations given in A.13.1.

- Traversed Paths. A trace of a program's execution can be provided.

- Name-Space Values. The dynamic state of all program name-space values which occupy primary memory and control memory can be provided.

- External System Status. Information on the status or states of all simulated external hardware and software as supplied to the abstract machine can be provided.

A.14 COMPUTER SIMULATORS

A computer simulator provides a lower level of abstract machine, as well as the environments of all abstract machines and can be used to monitor the next higher levels of abstract machines. The computer simulator is usually called the host computer and the abstract machines are called the target computer.

A.14.1 Abstract Machine Representation

This tool should provide program state information in the representation described for each of the following abstract machines:

- Micro-programmable computer. Program state information should be provided in terms of the values of flip-flops, data buses, registers, microinstruction addresses, end conditions, and symbolic functions.

- Basic Computer. Program state information should be provided in terms of machine register contents, location counters, memory addresses, and memory contents.

A.14.2 Capability and Usability Features

This tool should provide the following features:

- Instruction Simulation. A host computer can be programmed to operate the instruction set of the target computer by simulating its instruction execution. In doing this, the host computer can provide diagnostics on the instructions being operated.

- Network/External Environment Simulator. While simulating the target computer, the host computer may also be programmed to simulate interfacing, computers in a network system or the interfacing external environment. The network or environment simulator appears to the target computer software operating in the host computer as the real network or environment, complete with terminals, operators, sensor inputs, and interfacing hardware. For network simulation, the computer simulator generates all queries and responds to all answers by simulating the characteristics of the actual terminal or operators. For external environment simulation, the computer simulator provides all interfacing hardware inputs in the proper formats and at the proper formats and at the proper rates, and responds to all outputs of the target computer software.

A.14.3 Program State Information Requirements.

The program state information requirements for computer simulators are the same as for computer emulators (Section A.13.3).

A.15 GRAPHIC OUTPUT

Graphic output provides a higher level of program state representation than that associated with any abstract machine described in this report. This technique can be accomplished by analysis and evaluation software which transform the program state information generated by other tools and produces a new representation that is easier to comprehend by the debugging analyst. Graphic display systems are, of course, the primary means of presenting graphic output. Other devices can be used for most of the same forms of graphical representation, however, and in some cases are better suited for the representation. These devices include point plot display systems, alphanumeric terminals, oscilloscopes, and printers.

A.15.1 Abstract Machine Representation

Abstract machine representation is not applicable to this tool. Program state information is provided at a higher level than any abstract machine.

A.15.2 Capability and Usability Features

This tool should transform program state information into the following analogous representations:

- Analogous Images. The form of analogous images should be closely related to their meaning or purpose. An example is a model of a simulated musical instrument: a time-sequenced graph representing the dynamic evolution of amplitude, pitch variation, and tonal range is easier to comprehend than a table of numbers. For debugging purposes, the following forms of analogous images are applicable:
  - Plots. An instruction location versus number-of-instructions plot can be used to indicate areas of excessive timing for debugging timing problems.
  - Dynamic Flowcharts. Dynamic flowcharts show the structure of a program during a particular operation and provide a quick way of isolating problems due to bounds violations.
  - Bar Graphs. Bar graphs can be used to display the number of times each instruction, or interval of instructions, has been executed. This also is useful for debugging timing problems.
- Symbolic Representation. Symbolic images stand for concepts that are too abstract to analogize, such as numbers, algebraic and logical terms, and the characters and words that constitute language. Symbolic representations are particularly useful because they provide a means of handling concepts that are difficult to portray directly, such as generalizations and abstract relations. For debugging purposes, the following forms of symbolic representation are applicable.

- Tree-Like Flow Analysis. A tree-like representation of the flow of data into a particular segment or statement of code, showing previous value settings, can make it quickly apparent where the source of an error lies. Also, the representation can be used to show the dynamics of execution of a program, with the nodes of the tree corresponding to the code being executed brightening as execution moves from one node to another.

- Equation Representation. Mathematical representation of realworld phenomona, such as a model of a satellite's orbit, can be pictorially represented by two or three dimensional structures.

- Interface Representation. The interfaces between segments of code, programs, and subsystems can be shown symbolically by interface flow diagrams. The dynamic performance of a certain interface can be depicted by a diagram of the interface and a dynamic representation of throughput and loading activity being performed as the system operates.

A.15.3 Program State Information Requirements

The following components of program state information can be represented by graphic output.

- Traversed Paths. Representations such as a tree-like flow of a program, or flowcharts, provide information on traversed paths.

- Name-Space Values. The dynamic state of name-space values which occupy primary memory can be provided.

- External System Status. Representations of external hardware and software and the flow of status, control, and data through the interface, can be provided.

- Simultaneous Events. A plot can be presented showing simultaneous events by the intersection of the graphs of two independently operating functions.

- Space Allocation and Timing. The space allocation and relative timing information can be represented by plots and graphs.

A.16 SOFTWARE BREAKPOINTS/TRAPS

Software breakpoints/traps, which include unconditional, conditional, and frozen state breakpoints/traps, operate within the abstract machine and provide a means of detecting, and subsequently altering, various program state conditions. They specify a particular action to take place when a specified location is reached or a specified condition is satisfied.

A.16.1 Abstract Machine Representation

This tool should provide program state information in the representation described for each of the following abstract machines:

- Basic Computer. Locations should be specified in terms of absolute locations; conditions in terms of registers, memory contents, and operation codes; and actions in terms of machine instructions.

- Operating System. Locations should be specified in terms of operating system program calls and data block module names; conditions in terms of before or after scheduled events, I/O events and interrupt events; and actions in terms of operating system functions.

- Assembler. Locations should be specified in terms of entry point, statement numbers or labels and data names; conditions in terms of symbolic operation codes and data names; and actions in terms of labels, modifiers, data names and data values.

- Macro Library. Locations should be specified in terms of source language entry point, statement, and data names; conditions in terms of before or after macro-calls; and actions in terms of macro-name and input/output parameters.

- Compiler/Interpreter. Locations should be specifiable in terms of HOL statement number and labels and data names; conditions in terms of sequential operators (IF, IFEITH, ORIF, ELSE), logical operators (AND, OR, NOT), relational operators (EQ, GR, GQ, LQ, LS, NQ), and arithmetic operators (addition, subtraction, multiplication, division, exponentiation); and actions in terms of assignment operator (=), exchange operator, and sequential operators (FOR, DO DOWHILE, GOTO, STOP, RETURN).

- Subroutines. Similar to macro-library representation.

A.16.2 Capability and Usability Features

This tool should provide the following features:

- Location/Reference Breakpointing/Trapping. The capability to interrupt program execution when a specified location is encountered in a program's execution should be provided. The specification should be in terms of the abstract machine.

- Conditions. For conditional breakpoints/traps, the capability to specify a logical expression to be evaluated when the breakpoint/trap location or reference is encountered should be provided. The logical expression should be true in order for the tools' "action" to occur. The specification of conditions should be in terms of the abstract machine.

- Actions. For unconditional breakpoints/traps, and for conditional breakpoints/traps whose conditions have been met, a specified action should result. The actions should include the capability to display instructions or data, modify instructions or data, and to affect program control (e.g., resume execution at a new location). The specification of actions should be in terms of the abstract machine.

- Frozen State. In this mode, breakpoints or traps result in the program execution being suspended in a status that exactly reflects the result of all machine instructions executed prior to encountering the breakpoint or trap. Other programs inside the computer still continue to run independently of the frozen program; the frozen state applies to a program, not to the computer itself. It can be controlled in several ways depending upon the computing environments, and whether the system is interactive or batch.

- Parallel Processing. For systems involving parallel processing (multiprocessors, array processors, associate processors, pipeline processors, and distributive processors), the capability should exist to employ breakpoints/traps on each processor.

A.16.3 Program State Information Requirements

This tool should provide the following components of program state information in the abstract machine representations given as the result of "actions" in A.16.1.

- Symptom Locality. This tool provides the means of determining where a specified program state condition exists.

- Traversed Paths. This tool indicates whether a particular segment, a particular statement or location of a program is executed.

- Name-Space Values. This tool provides selective and conditional external hardware and software which are available during a program's execution is provided.

- Simultaneous Events. This tool can provide information on simultaneous events. For example, both processors of a parallel processor computer can be breakpointed/trapped to detect conditions where simultaneous location use or references are made.

- Space Allocation and Timing. No information provided.

A.17 HARDWARE BREAKPOINTS

Hardware breakpoints operate on the basic computer abstract machine and provide a means of detecting, and subsequently altering, various basic computer program state conditions. A particular location can be specified that when reached or referenced causes a specified action to take place.

A.17.1 Abstract Machine Representation

This tool should provide program state information in the representation described for the Basic Computer. Locations should be specified in terms of absolute locations and actions specified in terms of manually input machine instruction or memory contents at absolute locations.

A.17.2 Capability and Usability Features

This tool should provide the following features:

- Location Breakpoint. A location breakpoint is accomplished by wiring a comparator module to a control switch panel. This hardware matches the contents of the memory address register with the contents of control switches in order to halt the computer at the selected breakpoint address.

- Actions. After a hardware breakpoint has resulted in a halt to computer execution, a specified action should result. The action specified occurs at the time of the breakpoint, and is made in terms of the basic computer abstract machine, usually via the computer operator's console.

- Parallel Processing. For systems involving parallel processing (multiprocessors, array processors, associate processors, pipeline processors, and distributive processors), *the capability should exist for breakpointing each processor independently.*

A.17.3 Program State Information Requirements

This tool should provide the following components of program state information in the abstract machine representations given as a result of the "actions" specified in A.17.1.

- Traversed Paths. This tool indicates whether a particular segment of code or a particular location is executed.

- Name-Space Values. This tool can be used to provide the static state of specified program name-space values which occupy primary memory at a particular point in a program's execution.

- External System Status. The interface information of all external hardware and software which are available at a particular point in a program's execution can be provided.

- Simultaneous Events. This tool can provide information on simultaneous events. For example, both processors of a parallel processor computer can be breakpointed via hardware to detect simultaneous location reference.

A.18 REVERSIBLE EXECUTION/BACKTRACKING

Reversible execution/backtracking tools operate within the abstract machine and provide the traversed paths of the abstract machine which lead to a particular program state.

A.18.1 Abstract Machine Representation

The abstract machine representation for this tool is the same as for dynamic internal traces and recorded traces (Sections A.7.1 and A.8.1).

A.18.2 Capability and Usability Features

This tool should provide the following features:

- Backtrack Limits. The backtrack capability is based on a storing of information associated with a certain number of program instructions as the program is operating. A simulated execution of these instructions can be performed in reverse order. The number of instructions for which information is saved before old data is replaced by new data is a function of the space vailable for this tool.
- Jump Recording. To help hold down the amount of data which has to be stored for later replay, it is possible to record data only at places where jumps occur, or to record only the jumps themselves, plus more infrequent program state recording.
- Forward Trace Capabilities. In replaying the stored information, this tool should provide all of the capabilities stated in Section A.8.2. for recorded traces.
- Backtrack Operation Specification. The capability should exist of specifying a retrace until a statement or location is encountered or until a condition (e.g., a logical expression) becomes true. The specification should be in terms of the abstract machine and should be similar to the conditions specification applicable to software breakpoints/traps (Section A.16.1).

A.18.3 Program State Information Requirements

The program state information requirements for this tool are the same for dynamic internal traces (Section A.7.3).

A.19 INTERACTIVE MODIFICATION TOOLS

Interactive modification tools operate within the abstract machine and provide a means of dynamically altering the program state of the abstract machine.

A.19.1 Abstract Machine Representation

This tool should provide program state information in the representation described for each of the following abstract machines:

- Basic Computer. Modifications at this level are in terms of memory locations, machine instructions and memory data contents.

- Operating System. Modifications at this level are in terms of operating system functions.

- Assembler. Modifications at this level are in terms of assembly language labels, operation codes, modifiers, data or operand names, and data values.

- Macro Library. Modifications at this level are in terms of macro name and input/output parameters.

- Compiler/Interpreter. Modifications at this level are in terms of statement and data names, assignment operator (=), exchange operator, and sequential operators (FOR, DO, DOWHILE, GOTO, STOP, RETURN).

- Subroutines. Modifications at this level are in terms of subroutine name and input/output parameters.

A.19.2 Capability and Usability Features

This tool should provide the following features:

- Setting Data. To set data, the name or location of the data should be specified in terms of the abstract machine and the data values entered should be assumed to be in units equal to those declared in the abstract machine.

- Setting Instructions. To set instructions, the location or statement and the instruction should be specified in terms of the abstract machine and associated directives should exist to allow instructions to be added, deleted, changed, or replaced.

- Altering Execution Flow. To alter execution flow, the location should be specified in terms of the abstract machine. The actual branching should result from specification in terms of the abstract machine or from a command (such as BRANCH TO) meaningful to the debugging tool.

- Conditional Settings. A conditional setting feature can exist which requires that certain program state conditions exist before the setting or branching action takes place. The condition specification (e.g., a logical expression) should be in terms of the abstract machine and be similar to the conditions specification applicable to software breakpoints/traps (Section A.16.1).

- **Interactive Commands.** Where possible for an abstract machine, the interactive debugging system should allow modifications to be initiated via user-oriented commands, such as: SET, CALL, AT, DELETE, ADD, REPLACE, BRANCH TO, LOAD, STORE, BEFORE, AFTER, GO, STOP, ON CALL, and CLEAR.

A.19.3 Program State Information Requirements

This tool should provide the following components of program state information in the abstract machine representations given:

- Traversed Paths. The effect of altering paths traversed can be observed.

- Name-Space Values. The effect of altering name-space values can be observed.

- External System Status. The effect of altering information provided by external hardware and software can be observed.

A.20 RECOMPILATION/CORRECTION

Recompilation/correction methods are used on programs written in the abstract machine language and provide a means of modifying the program state of the abstract machine so the effects can be observed.

A.20.1 Abstract Machine Representation.

This tool should provide program state information in the representation described for each of the following abstract machines:

- Basic Computer. Octal, hexadecimal, or symbolic correctors may be used to add, insert, change, or delete instructions and/or data to the basic computer.

- Assembler. Recompilation can be performed by the assembler.

- Macro Library. Recompilation can be performed in the original source language.

- Compiler/Interpreter. Recompilation can be performed in the HOL compiler.

- Subroutines. Recompilation can be performed in the original source language.

A.20.2 Capability and Usability Features

This tool should provide the following features:

- Recompilation. The system's assembler or compiler should possess an update capability where the existing source code can be changed only where necessary and a program recompiled.

- Partial Recompilation. For interactive systems, the capability to recompile or reassemble selective portions of a program's source code should be considered. This allows for faster modification to occur, and in some cases, the modification time is transparent to the user in a manner similar to interactive modification tools (Section A.19).

- Correctors. Absolute or symbolic corrector provides a means of temporarily modifying a program and its data.

- Data Base Updates. A user-defined input capability should exist for updating a data base. The user should be able to pre-define the input formats to be used for individual data blocks of the data base. This capability should include the ability:

  - To set the length of a table.

  - To input a string of data, based on data block structure, without specifying each associated data identifier.

  - To clear (zero) a table and array.

  - To provide unit conversion algorithms so external data units can differ from units used internally.

  - To insert, add-on, delete, delete, replace, exchange, or modify entries of a table.

  - To repeat a substring of data for a table when a table item is to be set with a pattern of data.

A.20.3 Program State Information Requirements

This tool should provide the following components of program state information in the abstract machine representations given in A.20.2.

- Traversed Paths. The effect of altering paths traversed can be observed.

- Name-Space Values. The effect of altering name-space values can be observed.

A.21 HARDWARE MODIFICATION TOOLS

Hardware modification tools operate outside the abstract machine and provide a means of statically or dynamically altering the program state of a subset of the abstract machines, i.e., the micro-programmable and basic computer levels.

A.21.1 Abstract Machine Representation

This tool should provide program state information in the representations described for each of the following abstract machines:

- Micro-programmable Computer. Specification for modification can be in terms of flip-flops, data bus, micro-instruction address, registers, and symbolic functions.
- Basic Computer. Specification for modifications can be in terms of machine register contents, location counters, memory address and memory contents.

A.21.2 Capability and Usability Features

This tool should provide the following features:

- Setting Data. This tool should allow the components of the abstract machine to be set either statically when the machine is not executing or dynamically during execution.
- Instruction Insertion and Execution. This tool should allow a machine instruction at a specific location to be altered and then executed, or to cause branching to a new location for execution.
- Halting Execution. The tool should allow for halting of execution of the machine at a specific location.

A.21.3 Program State Information Requirements

The program state information requirements for this tool are the same as for recompilation/correction tools (Section A.20.3).

A.22 PROGRAM EXECUTION-ORIENTED RECORDING/REDUCTION

Program execution oriented recording reduction tools operate within the abstract machine and provide dynamic program state information at the abstract machine levels or a user-defined level.

A.22.1 Abstract Machine Representation

The recording function of this tool is performed at the basic computer level, while reduction/display is presented at any abstract machine level, except the micro-programmable computer level, including a completely user-defined level. The reduction/display representation for the various abstract machine levels is as follows:

- Basic Computer. Displays should be presented in terms of absolute locations, machine register contents, location counters, memory address of instruction obeyed, and memory contents.

- Operating System. Displays should be in terms of operating system elements operated and their input and output values.

- Assembler. Displays should be in terms of assembly statement labels, instructions executed, operands, newly assigned values, outcome of comparisons, and input/output values.

- Macro Library. Displays should be in terms of the macros operated and their input and output values.

- Compiler/Interpreter. Displays should be in terms of HOL source statement numbers executed, HOL instruction statement labels executed, or the HOL statements themselves, as well as the values of variable and function procedures required for expression evaluation. Also, any newly assigned values, the outcome of conditional tests, procedure calls, and the correspondence between formal and actual parameters can be presented.

- Subroutines. Displays should be in terms of the subroutines operated and their actual input and output values.

A.22.2 Capability and Usability Features

This tool should provide the following features:

- Common Recording Routine. A common recording routine can be employed, together with the trace interrupt mode capability, to record information on each instruction operated within user-specified bounds.

- Tracing. Capabilities similar to dynamic internal traces can be provided. (Section A.7.2).

- Flowback Analysis. Capabilities similar to reversible execution/backtracking can be provided (Section A.18.2).

- User-Defined Output Formats. A capability similar to Auxiliary Storage/Utility dumps capability of providing for user-defined display formats can be provided (Section A.6.2).

- Graphic (Scrolling) Display. The CRT can be used to show newly assigned values at the bottom of the screen while moving earlier assigned values up the screen. A composite view of how a variable changes throughout program execution can thus be provided.

- Source Code Execution. The source code statements being executed can be displayed by the use of a compiler generated symbol table to convert machine code to source code.

A.22.3 Program State Information Requirements

The program state information requirements of this tool are the same as for dynamic internal traces (Section A.7.3).

A.23 INPUT/OUTPUT-ORIENTED RECORDING/REDUCTION

Input/output-oriented recording/reduction tools operate within the abstract machine and provide program state information on all data transfers into and from the abstract machine. This tool records exactly what data was received and what data was sent by the program. It is primarily used for duplication of problem situations. It can be used to record what tests were run and the results that were generated. Manually prepared inputs to the abstract machine can be processed by the Test Input List tool discussed in Section A.24.

A.23.1 Abstract Machine Representation

The recording function of this tool is performed at the basic computer level while reduction/display is presented at various abstract machine levels, or at a higher level. The reduction/display representation for the various abstract machine levels is as follows:

- Basic Computer. Displays should be in terms of absolute locations and the contents of the basic memory unit (e.g., byte, word).
- Operating System. Displays of data should be tagged with common declaration or compool labels.
- Assembler. Displays of data should be tagged with associated assembly language labels.
- Compiler/Interpreter. Displays of data should be tagged with associated HOL labels.

A.23.2 Capability and Usability Features

This tool should provide the following features:

- Common Recording Routine. A common recording routine should be considered which is called whenever a operating system I/O function is requested or entered to process input or output data. Such a common routine usually employs a directory containing information on how each specific I/O transfer is to be recorded (buffer area, length· rates, etc.).
- Raw Input Recording. The capability should exist to record the major raw inputs of the system, e.g., radar inputs or telemetry inputs. The total input should be recorded at the main input interface of the system. This capability is useful during integration-level debugging only if this type of input is used. This capability is more likely to be used during operational-level debugging.

- Network Input/Output. The capability to record computer-to-computer transfers such as recording in terms of the system interface definition should be provided.

- Recording/Reduction Selectivity. The capability should exist to control what is recorded/reduced, the rates of recording, and to specify conditions for recording/reduction. Options such as start and end time spans should be provided.

- User-Defined Output Formats. A capability similar to Auxiliary Storage/Utility dumps capability of providing for user-defined display formats should be provided (Section A.6.2).

- Time-Tagging. A common system clock should be used to time-tag the recorded data.

A.23.3 Program State Information Requirements

This tool should provide the following components of program state information in the abstract machine representations given in A.23.1.

- Name-Space Values. The static state of all program name-space values input to or output from the abstract machine are provided.

- External System Status. The status or state of all external hardware and software that input to or output from the abstract machine is provided.

- Simultaneous Events. Recording of simultaneous events is provided, but the effects of the timing perturbation of the tool itself must be considered.

A.24 TEST INPUT LIST TOOLS

The test input list tools operate both independently of and within the abstract machine and provide information which augments the reported test results information requirements (Section 2) needed by the debugging process model. The tools also provide a means of duplicating a software problem. It generates and/or records manually prepared input data, as opposed to automatically generated input data, which is recorded by data recording/reduction tools (Sections A.22 and A.23).

A.24.1 Abstract Machine Representation

Not applicable.

A.24.2 Capability and Usability Features

This tool should provide the following features:

- CRT Terminal Input Recording/Listing. For interactive systems, that use CRT terminal input, the system software which processes the terminal inputs should possess a recording capability to capture terminal input selections, values, and input times. A means should also exist for obtaining hardcopy of this data such that it may be attached to a Software Discrepancy Report.

- Test Deck Listing Program. For batch systems based on card input, a simple, stand-alone program should exist to read and list a test deck. The listing should be in such a form that it can be conveniently attached to a Software Discrepancy Report.

- Switch Action Input Recording/Listing. Inputs made via such devices as paged-overlayed keyboards, numeric and fixed function keyboards, as well as system-specific devices such as weapon-system switch action consoles, should be recorded and listed in the same manner as described for CRT terminal input.

- Script Tapes. A script tape is a tape containing sequenced test inputs which may be input to the computer system as an alternate to manual inputs. For example, paged-overlayed keyboard commands, together with command time, can be specified on a magnetic tape and input in place of the manual actions and processed by the same system input software. The software to generate the script tape should also provide a listing of the test inputs which can be conveniently attached to a Software Discrepancy Report.

A.24.3 Program State Information Requirements

Not applicable.

A.25 COMPARATOR TOOLS

Comparator tools operate both within and outside the abstract machine and provide for a comparison of different sets or versions of the same program state information for various abstract machines. These tools are to be used to verify that the correct results produced before a program correction was made are still being produced. The tools are also useful for isolating differences between a new version of an element of the system and an older version.

A.25.1 Abstract Machine Representation

This tool should provide program state information in the representation described for each of the following abstract machines:

- Basic Computer. Differences should be provided data in terms of primary or secondary storage locations and basic units of storage (words, bytes, etc.)

- Assemblers. Differences in assembly language programs should be provided in terms of corresponding assembly code labels, symbolic operations, modifiers, operands, and data values.

- Compiler/Interpreter. Differences in HOL programs should be provided in terms of corresponding HOL labels, operations, and data units.

- Subroutines. Not applicable.

A.25.2 Capability and Usability Features

This tool should provide the following features:

- Secondary Storage Comparisons. The capability should exist to compare the secondary storage devices of the system, such as tapes, diskpacks, cards, and paper tape. Such comparisons should be in terms of the type of information stored on the media. For example, comparisons of versions of program source or object code, master tapes, and versions of data bases, compools, correctors, and libraries should be available in terms of the appropriate abstract machine representation.

- Internal Comparisons. Quite often the same set of data is used at different places within a program or within a system or programs. A means to record and then compare these sets should be considered. Outputs usually are made only if differences are found (Comparison dumps).

- Source Identification. The comparator tools output should identify the sources being compared.

- Selectivity. The capability should exist to compare only selective portions of one or more sources. For example, specific records of a tape, specific data blocks of a data base, or specific regions of a program.

- Multiple Sources. The capability should exist to compare as many sources at one time as is appropriate for the device-type and information being compared.

- Offset Data Comparison. In the case where two sources differ, but before and/or after the areas of difference there is correspondence, the tool should attempt to re-syncronize the data being compared so re-correspondence is recognized.

A.25.3 Program State Information Requirements

This tool should provide the following components of program state information in the abstract machine representations given in A.25.1.

- Symptom Locality. This tool provides symptom locality possibilities by comparing a version containing a problem with one with which a software problem was not experienced.

- Name-Space Values. By comparing recorded data, this tool can provide information about differences in name-space values between different versions of a program.

- Space Allocation and Timing. Information on space allocation differences can be provided.

A.26 PROBLEM STATUS REPORTERS

Problem status reporter tools operate independent of the abstract machine and provide management information associated with a software problem.

A.26.1 Abstract Machine Representation

Not applicable.

A.26.2 Capability and Usability Features

This tool should provide the following features:

- Software Discrepancy Report. The primary input to this tool is the software discrepancy report. This information is described in Section 2.4.

- Interim Resolution Data. This data updates the software discrepancy report and provides the current views of the debugging analyst on the cause of a problem, and the current status of its resolution. This is the second form of input to this tool.

- Final Resolution Data. This data further updates the software discrepancy reports and interim reports information and provides information on the solution of the problem. This is the third form of input to this tool.

- Problem Status. As the status of a problem changes up until it is finally closed, the status is input to this tool.

- Listable Output. The tool should be able to list the entire software discrepancy report for each problem. It should contain the capability to obtain a complete or a selective listing of problems. For example, the capability to list by status (open, closed, etc.), responsible contractor, subsystem, or priority. The tool should also allow for the full text of the data from the forms to be listed or simply a summary of the problems.

A.26.3 Program State Information Requirements

Not applicable.

A.27 SYSTEM STATUS SUMMARY TOOLS

System status summary tools provide the configuration and status of the test, hardware and software configuration active at the time of a problem. The tool is important for problem duplication.

A.27.1 Abstract Machine Representation

This tool should provide information on the configuration of all abstract machines, if applicable.

- Micro-programmable Computer. The version of the firmware active should be provided.

- Basic Computer. Operative/inoperative equipment, computer system ID, identification of tape and disc status summary, and peripheral equipment status should be provided.

- Operating System. Version of operating system used, versions of program(s) executed, versions of test cases used, versions of correctors or modified programs active, ID of active data base and master tapes active should be provided.

- Assembler. Assembler version used should be provided.

- Macro Library. Library version used should be provided.

- Compiler/Interpreter. Compiler/interpreter version used should be provided.

- Subroutine. Version of subroutine library used should be provided.

A.27.2 Capability and Usability Features

This tool should provide the following features:

- Automatic Operation. This tool should automatically record the version identification of all.

- Manual Operation. Manual operation of this tool should be available via computer operator request, interactive request, or as specification in a batch job card input deck.

- Time Information. Output should include time of job.

- Output Format. The output should be kept to one page and be easily attached to a Software Discrepancy Report.

A.27.3 Program State Information Requirements

Not applicable.

A.28 CHECKPOINT TOOLS

Checkpoint tools operate within the abstract machine and provide a means for returning the program state of the abstract machine to the state that existed prior to or at the point of the occurrence of a software problem. The tools are important for problem duplication.

A.28.1 Abstract Machine Representation

Not applicable.

A.28.2 Capability and Usability Features

This tool should provide the following features:

- System Save/Restore. A system save/restore capability should exist to enable the program environment to be saved between operation of individual programs, at the time of a manual interrupt by the computer operator, or at the time of an abnormal termination. The saved environment should subsequently be able to be restored at this point. To do this, the contents of memory, general registers, program counters, disc and equipment status are saved on tape or disc storage.

- Data Base Save/Restore. A data base save/restore capability should exist to enable the active data base to be retrieved and stored in its most currently updated state. The saved data base should subsequently be able to be reloaded and used by the application software.

- Selective Data Base Save/Restore. The capability should exist to selectively retrieve elements of an active data base. For example, one or more data blocks may be saved and later restored. This capability is useful where it is known that only selective elements are involved in a problem and allows for a less time consuming application of this tool.

A.28.3 Program State Information Requirements

Not applicable.

APPENDIX B: GLOSSARY

CADSAT Computer-Aided Design and Specification Analysis Tool. The purpose of this tool is to describe the requirements for information processing systems and to record descriptions in machine-processable form. The major components of CADSAT are the User Requirements Language (URL) and the User Requirements Analyzer (URA) which can operate in an interactive computer environment.

CDR Critical Design Review. A formal technical review of the design as depicted by either the specification and/or flow diagrams. sufficiently detailed to enable the programmer to code, compile, and debug a computer program, to assure that design requirements have been met before coding begins.

Configuration Control

The systematic evaluation, coordination, approval or disapproval, and implementation of all approved changes in the configuration item after formal establishment of its configuration identification.

Configuration Item (CI)

An aggregation of hardware, computer programs, or any of its discrete portions, which satisfies an end-use function and is designated by the Government for configuration management. CI's may vary widely in complexity, size, type, from an aircraft, electronic, or ship system to a test meter or round of ammunition.

Configuration Management

A discipline applying technical and administrative direction and surveillance to (1) identify and document the functional and physical characteristics of a configuration item (2) control changes to those characteristics, and (3) record and report change processing and implementation status.

Correctness

The state in which the computer program behavior is in accord with all specifications.

CPC Computer Program Component. A functionally or logically distinct part of a computer program distinguished for purpose of convenience in designing and specifying a complex computer program as an assembly of subordinate elements.

CPC Code and Test

This phase begins with the Critical Design Review. All coding of CPCs is completed and each CPC is tested individually to ensure that it satisfactorily serves its intended purpose.

CPC Detail Design

This phase commences at the Preliminary Design Review. A well defined and logically organized software system is developed. Included are a preliminary design of the data base in parallel with the structure of CPCIs and definitions and programming of the CPCs.

CPCI — Computer Program Configuration Item. A computer programming end product whose development and subsequent modification is subject to configuration management.

CPCI Integration and Testing

CPCs are assembled into CPCIs and CPCI testing is performed to assure operation of the software system in accordance with performance specifications.

CPCI Part I Specification (also known as Development Specification, or Requirements Specification).

A document applicable to a CPCI which states all necessary requirements in terms of performance, interface, design constraints, functional characteristics, and the tests required to demonstrate achievement of those requirements. The initial draft of this document is known as the Preliminary Design Document.

CPCI Part II Specification (also known as Product Specification, or Design Specification)

A document which follows the CPCI Part I Specification (i.e., Development Specification) and which provides a detailed technical description of the CPCI. The Source Code Specification is included as part of the CPCI Part II Specification. The initial draft of the CPCI Part II Specification is known simply as Draft Part II Specification.

CPCI Qualification

Customer checks and approves the developed software system as satisfactory.

CPCI Test Plan

Based upon the requirements contained in the Part I Specification, the CPCI Test Plan identifies the PQTs and FQTs to be conducted. In addition, the CPCI Test Plan states the test objectives, relative schedules, and support requirements.

CPDP — Computer Program Development Plan. The CPDP is a plan that identifies the actions needed to develop and deliver computer program configuration items and necessary support resources.

CPT & E — Computer Programming Test and Evaluation. Tests conducted prior to and in parallel with preliminary or formal qualification tests. These tests are oriented primarily to support the contractor's design and development process.

Data Dictionary

A table that gives the attributes of data items.

Deadlocks

A condition where two processes are mutually waiting for a condition which will never occur.

Design Specification

Same as CPCI Part II Specification.

Detail Design Documents

Documents the developer's detail design of CPC modules to govern the development of computer program code. This document must be delivered in time for the Critical Design Review.

Development Specification

Same as CPCI Part I Specification.

Draft Part II Specification

See CPCI Part II Specification for explanation and definition.

ECP — Engineering Change Proposal. A plan which includes both a proposed engineering change and the documentation by which the change is described and proposed.

FCA Functional Configuration Audit. A formal audit to validate that the development of a configuration item (CI) has been completed satisfactorily and that the CI has achieved the performance and functional characteristics specified in the functional or allocated configuration identification.

FQT Formal Qualification Tests. Formal tests oriented toward testing of the integrated CPCI, normally using operationally configured equipment at the System Development Test and Evaluation site prior to the beginning of System Development Test and Evaluation.

GFE Government Furnished Equipment.

HAL/S A comprehensive programming environment including real time higher order programming language and compiler used for the development, debugging, and maintenance of avionics software.

Handbooks and Manuals

This is a category of documents which contain information about, or derived from the CPCI functions and design. Handbooks and manuals are to be used by system personnel responsible for use and support of the CPCI during systems operations.

Hierarchical Decomposition

Within the context of top-down design, hierarchical decomposition is the process of decomposing a design component at one level to a set of subcomponents at the next lower level of abstraction. This process is iterated until an implementation level of detail is reached.

HIPO Hierarchical Input Processing Output. A procedure orignated by IBM for the purpose of supplying a graphical representation of the inputs used by a program component, the processing performed, and the outputs produced. The processing is described in terms of lower level software components which are in turn represented by a HIPO diagram thus presenting a hierarchical picture of program structure.

HOL Higher Order Language. A machine-independent programming language in which the characteristics of a particular computer are not apparent.

Installation

A phase of the software development life cycle during which adaptation of the software to the special requirements of a specific installation is made.

IVV — Independent Verification and Validation. Verification and validation of a system performed by personnel not involved in the development of the system. (See definitions of verification and validation).

Listings — Listings refer to listings of computer program instructions and data generated in the process of developing the CPCI code.

Logical Scope

Identification of a functional area of a program and its subcomponents to be studied for correctness.

Modification ID

Appendage to the name of a software component that indicates the level of modification represented by the current state.

Module — Used in this document to describe any computer program unit that can be compiled or assembled. In this document, module is used as a synonym for terms such as CPC, CPCI, or submodule if no specific distinction between the above entities is crucial to the meaning of a statement.

Operation and Deployment

A phase of the software development life cycle during which the software system is used in its planned operational capacity.

PCA — Physical Configuration Audit. The formal examination of the "as built" configuration of a unit of a CI against its technical documentation in order to establish the CI's initial product configuration identification.

PDN — Program Design Notation. A formal language for representing program design.

PDR — Preliminary Design Review. A formal review of the preliminary design of a system functional area or of a configuration item to establish system compatibility of the design, identify specific engineering documentation, and define physical and functional interface relationships.

PQT — Preliminary Qualification Tests. Formal tests oriented primarily toward verifying portions of the CPCI prior to integrated testing/formal qualification tests of the complete CPCI. These tests will typically be conducted at the contractor's design and development facilities.

PPL — Program Production Library. A group of manual or automated procedures used to control and keep records of the developing software.

Preliminary Design

Initial software development phase following award of contract. During this phase, functional interfaces between CPCIs, between CPCIs and equipment (e.g., computer), and communication links (message formats and data transfers) are identified and described.

Preliminary Design Document

This document is issued at the Preliminary Design Review as a preliminary draft of the CPCI Part I specification. It documents the developer's design approach to the CPCI as a whole. (See CPCI Part I Specification for further explanation).

Product Specification

Same as CPCI Part II Specification.

Race Conditions

Occurs when two concurrent processes access share data in a non-interlock manner.

Requirements Specification

Same as CPCI Part I Specification.

SADT — Structured Analysis Design Tool. Top-down graphical system design technique developed by SOFTECH.

SDR — Software Discrepancy Report. A document containing the description of the test case and test results. This document is used to initiate the debugging process and to track the status of the error detection and resolution.

Source Code Specification

Part of the Part II CPCI Specification. The Source Code Specification documents the coding of the CPCI.

Source Language

Program language used by the programmer to code input to a compiler or assembler (e.g., PL/I).

SREM Software Requirements Engineering Methodology. A part of the Ballistic Missile Defense Advanced Technology Center (BMDATC) program directed toward improving the methodology used in the development of software for BMD programs. SREM is a part of this program and addresses requirements generation for software development. SREM is composed of a combination of languages, tools, and procedures designed to reduce or eliminate known error sources. Data processing subsystem requirements in the SREM approach for BMD are predicated on an input-processing-output orientation with processing flow through the subsystem being described in terms of required paths through the subsystem.

Strong Type Checking

An attribute of a programming language whereby the data type of every object in a program can be inferred at compile-time. Languages that have strong type checking rules permit the compiler to detect illegal operations on data.

System Integration Test

Entire software/hardware system is tested to demonstrate proper functioning.

System Operations

Operate or assist in operating system in prescribed manner which may include assuring the software system's continued efficiency and correctness, diagnosing deficiencies, or producing and incorporating improvements or corrections.

Test Procedures

Documents the procedures to be followed during a formal test. This document specifically states the test objectives, location and schedule, detailed scenario of test events, inputs, expected results, and methods of test data recording and analysis.

Test Reports

Documents the accomplishment of each formal test (i.e., FQT or PQT) including results, problems encountered and actions taken to resolve the problems.

URA User Requirements Analyzer. A software program which records in a data base a description of the requirements for an information processing system written in URL and performs analysis on the description of these requirements. (Part of CADSAT).

URL User Requirements Language. Language used for describing the requirements for an information processing system in a formal, machine processable format. (Part of CADSAT).

Validation

Comprises those evaluation, integration, and test activities carried out at the system level to assure that the final developed system satisfies the requirements of the System Specification.

Variable Scope Rules

A set of rules which define when a variable is visible to a program component.

Verification

The iterative process of determining whether the product of selected steps of the CPCI-development process fulfills the requirements levied by the previous step.

Virtual Machine

A "machine" created out of hardware and/or firmware and/or software that supports a family of programming applications.

APPENDIX C: BIBLIOGRAPHY

1. A Collection of Technical Papers; Computers in Aerospace Conference; Published by American Institute of Aeronautics and Astronautics; Los Angeles, California; 1977.

2. Finfer, M. and Mish, R.; Software Acquisition Management Guidebook: Software Cost Estimation and Measurement; System Development Corporation; TM-5772/007/02; 1978.

3. Miller, G. A.; Psychology Review; Vol. 63; pp. 81-87; 1956.

4. Mish, R., Munsow, J., Newlands, E.; Software Development Manual; System Development Corporation; TM-(L)-5589/000/00; 1976.

5. Neil, G.; Software Acquisition Management Guidebook: Reviews and Audits; System Development Corporation; TM-5772/006/02; 1977.

6. Neil, G.; Software Acquisition Management Guidebook: Software Quality Assurance; System Development Corporation; TM-5772/001/03; 1977.

7. Searle, L. V.; An Air Force Guide to Computer Program Configuration Management; System Development Corporation; TM-5772/005/01; 1977.

8. Software Acquisition Management Guidebook: Verification; System Development Corporation; TM-5772/002/02; 1977.

9. Stanfied, J. and Skrukrud, A.; Software Acquisition Management Guidebook: Software Maintenance; System Development Corporation; TM-5772/004/02; 1977.

10. Strukrud, A. and Willmorth, N.; Software Acquisition Management Guidebook: Validation and Certification; System Development Corporation; TM-5772/003/02; 1977.

A more comprehensive bibliography is contained in Volume III.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.